程序员的数学思维修炼（趣味解读）

周颖　等编著

一本专门为程序员而写的数学书，训练数学思维，增强职场竞争力

书中没有罗列晦涩难懂的数学公式和推导，而是代之以生动有趣的数学实例

读者不必精通高深的数学知识，只需要具备四则运算和基本的逻辑思维即可阅读

通俗易懂，风格活泼，趣谈110个实例，并给出了33个具体的程序代码实现



清华大学出版社

程序员的数学思维修炼

（趣味解读）

周颖　等编著

清华大学出版社
北　京

内 容 简 介

本书是一本专门为程序员而写的数学书，介绍了程序设计中常用的数学知识。本书门槛不高，不需要读者精通很多高深的数学知识，只需要读者具备基本的四则运算、乘方等数学基础知识和日常生活中的基本逻辑判断能力即可。本书拒绝枯燥乏味的讲解，而是代之以轻松活泼的风格。书中列举了大量读者都很熟悉，而且非常有趣的数学实例，并结合程序设计的思维和算法加以剖析，可以训练读者的数学思维能力和程序设计能力，进而拓宽读者的视野，增强职场竞争力。

本书共 11 章，分别介绍了数据的表示、神奇的素数、递归、排列组合、用余数进行数据分组、概率、复利、数理逻辑、推理、几何图形构造、统筹规划等程序设计中常用的数学知识，从而引导读者深入理解编程中的数学方法和思路。

本书适合广大程序设计人员及数学爱好者阅读，尤其适合有一定程序设计经验，但还需要进一步加深对程序设计理解的人员阅读。本书对 IT 求职人员、信息学竞赛和大学生程序设计竞赛等参赛学员也有很好的参考价值。

图书在版编目（CIP）数据

程序员的数学思维修炼（趣味解读）/ 周颖等编著. —北京：清华大学出版社，2014
ISBN 978-7-302-35060-6

Ⅰ. ①程…　Ⅱ. ①周…　Ⅲ. ①电子计算机－数学基础　Ⅳ. ①TP301.6

中国版本图书馆 CIP 数据核字（2014）第 006758 号

责任编辑：夏兆彦
封面设计：欧振旭
责任校对：胡伟民
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张：**19.75　　**字　　数：**496 千字
版　　次：2014 年 4 月第 1 版　　**印　　次：**2014 年 4 月第 1 次印刷
印　　数：1～ 000
定　　价：　　元

产品编号：056332-01

前　言

数学在人类文明的发展过程中起着非常重要的作用。数学推动了重大的科学技术进步。从远古的“结绳记事”，到现代计算机技术的快速发展，都与数学这门学科的发展密不可分。

无论是日常生活中简单的商品交易计算，还是神舟飞船设计中复杂的计算，都离不开数学。生活即数学。没有二进制，就不会有现在的计算机；没有几何学，就没有现在的高楼大厦……。

对于程序员来说更需要知道：数学是计算机科学的基础。在我国，绝大部分大学的计算机科学系都是从数学系中分出来的。由此也可以看出，计算机科学与数学的关系非常紧密。

数学是一门化繁为简的学科。通过数学，可以对现实生活中的很多不同事物进行高度抽象，从而能找出不同事物的共性。不过，由于数学的这种高度抽象，又使数学变得很难学。特别是一些复杂的公式推导，看起来就头痛。

本书面向程序员介绍了程序设计中常用的数学基础知识。通过阅读本书，可以训练程序员的数学思维能力和程序设计能力，进而拓宽视野，增强职场竞争力。

本书特点

- **简单易懂**　用通俗易懂的语言讲解知识点，尽量避免复杂的公式推导过程，让读者能够轻松阅读并掌握相关的数学知识。
- **门槛很低**　阅读本书的读者不需要精通很多高深的数学知识，只需要具备基本的四则运算、乘方等数学基础知识和日常生活中的基本逻辑判断能力即可。
- **生动有趣**　本书拒绝枯燥乏味的讲解，而是代之以轻松活泼的风格，讲解时列举了大量我们都很熟悉，而且非常有趣的数学实例。
- **内容丰富**　本书从最简单的数据的表示开始，对素数、递归、排列组合、逻辑推理、几何构造、统筹规划等方面都会逐一介绍，涵盖了程序员需要掌握的数学知识。
- **图文并茂**　讲解每个知识点和实例时，都给出了简单易懂的图示和必要分析，让读者理解起来清晰明了，没有任何障碍，也让读者感觉到学习数学并不困难。

本书内容概述

第 1 章通过一则童话故事导入了数据大小的知识，然后逐步介绍了十进制、二进制、八进制、十六进制以及其他常用进制的知识，还介绍了不同进制的转换方法。

第 2 章从素数的判断开始，逐步介绍了与素数相关的数学知识，包括孪生素数、梅森素数、哥德巴赫猜想、RSA 的应用等内容。

第 3 章介绍递归这种自己调用自己的方法，通过阶乘、汉诺塔、斐波那契数列等经典实例，练习从复杂事物中发现递归结构的方法。

第 4 章的主题是排列组合，从乘法原理、加法原理入手，介绍了排列与组合的概念和关系，并研究了计算机中的字符编码、密码长度等相关内容。

第 5 章讨论余数。主要介绍使用余数对数据进行分组，如日历、一些小魔术都是通过余数分组的规则进行的；本章还讨论了计算机中的奇偶校验及两个有趣的问题（座位安排和智叟分牛）。

第 6 章介绍概率的相关知识，首先从两个常见的事例导入概率的概念，接着从军事故事、赌场游戏、中奖概率等方面介绍了概率的实际应用。

第 7 章学习翻番的知识。首先介绍翻番和翻倍的概念、计算方式。接着进一步通过复利的威力、对折纸张、舍罕王的赏赐等实例，展示了翻番这个令数据快速增长的数学概念。最后还介绍翻番的逆运算——折半的应用。

第 8 章学习数理逻辑的相关知识，介绍了逻辑、命题逻辑、布尔逻辑、逻辑的重叠与遗漏等概念，最后介绍了通过卡诺图化简逻辑表达式的方法。

第 9 章则在第 8 章的基础上进一步讨论了逻辑推理，包括演绎推理中的三段论、选言推理、假言推理、关系推理，以及归纳推理中的完全归纳推理和不完全归纳推理。

第 10 章介绍了几何图形构造的基础知识，从花盆摆放、残缺棋盘、丢失的线条等有趣实例，初步了解几何图形构造，最后还介绍了几何图形的分割与拼接。

第 11 章讨论统筹规划相关知识，首先从田忌赛马这个古老故事中看出统筹规划的重要性，然后通过生活中的两个简单例子认识统筹规划，最后还讨论了“背包问题”及其程序设计方法。

本书读者对象

本书可适用以下各类人员阅读：

- ❑ 计算机专业的学生；
- ❑ 数学专业的学生；
- ❑ 程序设计人员；
- ❑ 数学爱好者；

- 编程爱好者。

本书作者

本书由周颖主笔编写。其他参与编写的人员有韩先锋、何艳芬、李荣亮、刘德环、孙姗姗、王晓燕、杨平、杨艳艳、袁玉健、张锐、张翔、陈明、邓睿、巩民顺、吉燕、水淼、宗志勇、安静、曹方、曾苗苗、陈超。

编写本书的过程中，虽然编者竭尽全力，不敢有丝毫疏忽，但恐百密一疏，书中仍难免存在不足之处，望广大读者批评指正。

编著者

目　　录

第 1 章　数据的表示

数学古称算学，是中国古代科学中一门重要的学科。根据中国古代数学发展的特点，可以分为 5 个时期，分别是萌芽、体系的形成、发展、繁荣和中西方数学的融合。

在数学的不同发展阶段，对于数据的表示都有一些不同的形式。从远古的结绳记数，到现在用计算机等现代科技设计记数，数的表示形式也在逐步演化。

本章主要介绍数据的各种表示形式，包括各种进制及进制之间的转换。

1.1　一则童话

根据我们所学的知识可知道，数据通常是用 0、1、2、3、4、5、6、7、8、9 这些数来表示，由这些数的不同组合表示现实生活中各种各样的数据。首先来看这个数列中的前两个数：0 和 1，从通常意义来说，0 就是什么也没有，真的是这样吗？对程度员来说不应该这样理解。

先来看这样一个问题，0 和 1 谁大？

0>1?　　0<1?

1.1.1　0 和 1 的故事

在数学王国里，胖子 0 与瘦子 1 常常为了谁大而争执不休。瞧！今天，这两个小冤家狭路相逢，彼此之间又展开了一场舌战。

瘦子 1 抢先发言：“哼！胖胖的 0，你有什么了不起？就像 100，如果没有我这个瘦子 1，你这两个胖 0 有什么用？”

胖子 0 不服气了：“你也甭在我面前耍威风，想想看，要是没有我，你就只是一个光杆呢？”

“哟！”1 不甘示弱，“你再神气也不过是表示什么也没有，看！1+0 还不等于我本身，你哪点儿派得上用场啦？”

“去！1×0 结果也还不是我，你 1 不也同样没用！”0 针锋相对。

“你……”1 顿了顿，随机应变道，“不管怎么说，你 0 就是表示什么也没有！”

“这就是你见识少了。”0 不慌不忙地说，“你看，日常生活中，气温 0 度，难道是没有温度吗？再比如，直尺上没有我作为起点，哪有你 1 呢？”

“再怎么比，我始终比你大。”1 信心十足地说。

听了这话，0 更显得理直气壮地说：“嘿嘿，你的大小还得我说了算，我站你左边，你就成 0.1，我站你右边你就是 10。怎么样？我可让你放大 10 倍，也可让你缩小 10 倍！”



眼看着胖子 0 与瘦子 1 争得脸红耳赤，谁也不让谁，一旁观战的其他数字们都十分着急。

这时，9 灵机一动，上前做了个暂停的手势：“你俩都别争了，瞧你们，1、0 有哪个数比我大？”

“这……”胖子 0、瘦子 1 哑口无言。

这时，9 才心平气和地说：“1、0，其实，只要你们站在一块，不就比我大了吗？”

1、0 面面相觑，半晌才搔搔头笑了。“这才对嘛！把自己的位置放正，就能起到应有的作用”。9 语重心长地说。

从以上故事可看出以下两点：

- 0 并不表示什么都没有。
- 数的大小与所处的位置有关系。

下面就来讨论这两个问题。

1.1.2 0 是什么都没有？

通常意义上，0 表示“没有”的意思。例如，“2012 年过去了，可我的收获为零！”这就表示在 2012 年没有收获。

但是，0 真表示什么都没有吗？

其实，0 不仅表示什么都没有，它还有更丰富的内涵。例如，0 度并不是没有温度，而是表示温度为 0 度，比零下 1 度高，比 1 度低，如图 1-1 所示。

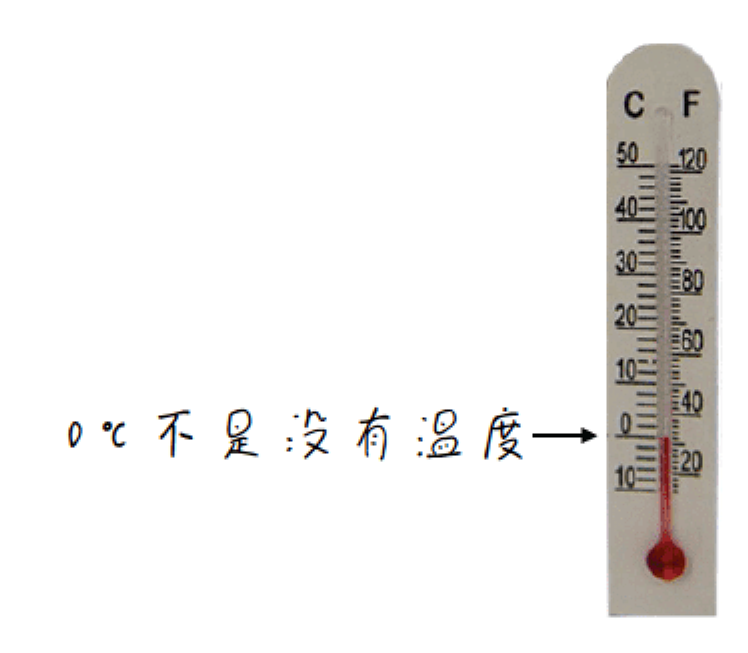


图 1-1

在日常生活的常用语中，也有很多用 0 来表示的，如“很多女孩子都喜欢吃零食”，这里的“零食”并不是表示没有“食”，如图 1-2 所示。



图 1-2

“为了增加收入，改善生活，很多程序员在业余时间都会接点零活来做。”这里的“零活”并不是没有“活”。

其实，在数学上，0 也并不是表示没有。例如，8 和 8.0 相等吗？其含义相同吗？

$$8 \neq 8.0$$

看起来在小数点后添加一个 0 是没有意义的，不过，其含义实际是不相同的。在近似数表示中，数字 8 表示数据只精确到个位，如 7.9、8.2 等数精确到个位都表示为 8。而 8.0 表示数据精确到十分位，如 8.02、7.99 等数精确到十分位都表示为 8.0。所以，从这个角度来看，8 和 8.0 是不相等的。

1.1.3　0 的位置

从“0 和 1 的故事”可看出，当 0 所处的位置不同时，其含义也不一样。如前面说的 8 和 8.0，当把 0 放在小数点后面时，从绝对值方面来看，两个数是相等的，但从近似数来看，小数点后多了一个 0，其表示的含义也就不一样了。

那么，在小数点左侧添加 0 呢？如果在数的最左侧添加 0，无论添加多少个 0，数的大小都不变。

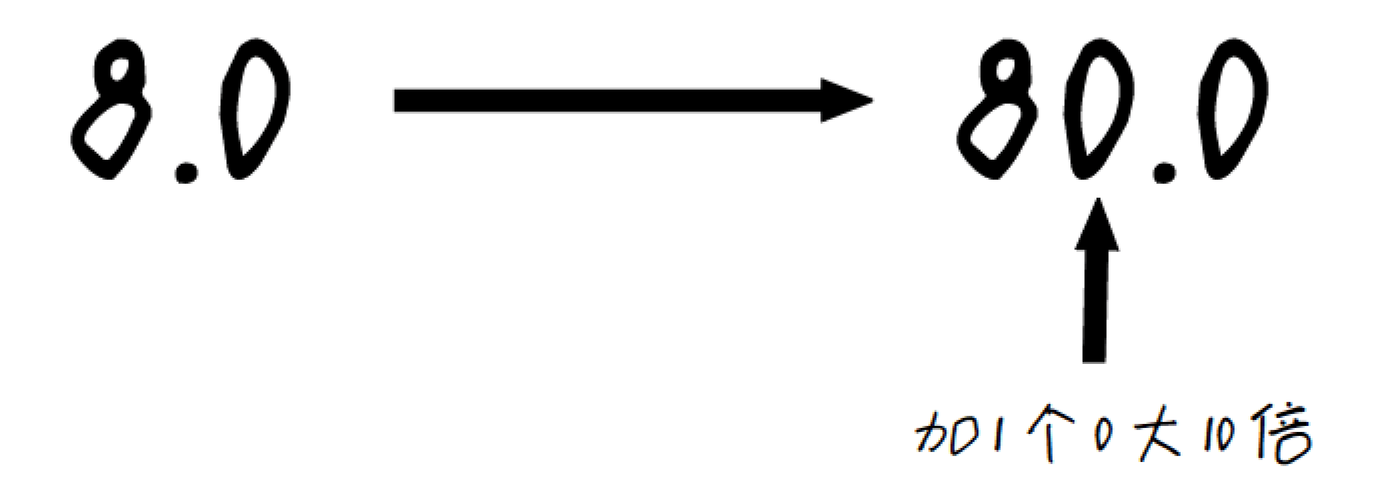
$$8=08 \qquad 80=080$$

但是，如果在数的中间插入 0，数的位置与数的大小关系就很明显了，如在 18 的中间插入一个 0，得到的是 108，很明显，其大小差别很大。

$$18 \neq 108$$

对于 18，表示十位为 1，个位为 8，也就是说，表示 18 这个数有 1 个 10，8 个 1。而 108，表示百位为 1，十位为 0，个位为 8，即表示有 1 个 100，0 个 10，8 个 1，这时的 0 是一个占位符，把 1 从十位挤到百位。

而如果在紧邻小数点的左侧添加 0，则数据会扩大 10 倍。



1.1.4 程序中的 0

在电子技术中，0 一般表示低电平，1 为高电平。在逻辑计算中，0 一般表示逻辑假（False），1 为逻辑真（True）。在数值运算中，0 与平常数学中 0 的含义相同。

在程序中，数据 0 有什么含义呢？

1. 未赋值的变量为0？

在不同的程序设计语言中，对于未赋值变量的处理不一样。

对于 Basic 类的程序语言，如 QB（Quick Basic，简称 QB）、VB（Visual Basic，简称 VB），如果数值型变量未赋初值，则其初始值为 0。例如，有以下 VB 程序代码：

```
Private Sub Test
    Dim i As Integer
    MsgBox "变量 i=" & i, , "变量初始值"
End Sub
```

在以上 VB 代码中，声明了变量 *i*，但未对其进行赋值。虽然未进行变量赋值初始化，但 VB 编译器会自动将这类数值型变量初始化为 0。因此，执行以上代码将显示如图 1-3 所示的对话框。

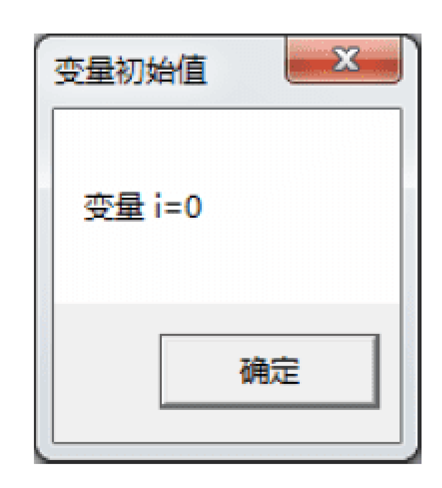


图 1-3

对程序员来说，VB 对变量进行初始化的方式很讨人喜欢，变量声明后就可以使用。但是，在.Net Framework 中，其处理方式又不相同，例如，以下是 VB.NET 中的代码：

```
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs)
_
    Handles Button1.Click
    Dim i
    MsgBox("变量 i=" & i, , "变量初始值")
End Sub
```

以上代码并不会出错，但运行后得到的结果如图 1-4 所示。从这个结果可看出，在 VB.NET 中，如果变量使用之前未进行初始化，这时其值为空（并不为 0）。

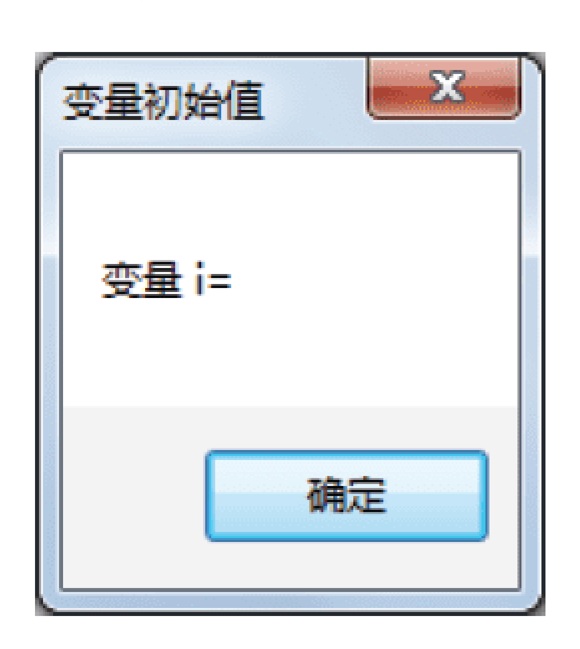


图 1-4

其实，在 Visual Studio 开发环境中仔细观察代码，可看到在 MsgBox 函数中的变量 *i* 下方有一个波浪线，将鼠标指针指向变量 *i*，可看到如图 1-5 所示的提示信息，提示变量 *i* 在赋值前被使用。

```
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) _
    Handles Button1.Click
    Dim i
    MsgBox("变量 i=" & i, , "变量初始值")
End Sub
```

量 "i" 在赋值前被使用。可能会在运行时导致 null 引用异常。

图 1-5

对于 C 语言系列的程序设计语言（如 C、C++、C#等），程序员就没那么幸运了，未初始化的变量编译器并不会将其初始化为 0，而且不同编译系统可能会采用不同的处理方式。例如，有如下的 C#程序：

```
private void button1_Click(object sender, EventArgs e)
{
    int i;
    MessageBox.Show(string.Format("变量 i={0}", i), "变量初始值");
}
```

以上的 C#程序是没办法编译通过的。在 Visual Studio 开发环境中可以看到变量 *i* 下方有一条波浪线，将鼠标指针移到变量 *i* 上，可看到如图 1-6 所示的错误提示信息，提

示使用了未赋值的局部变量 *i*。

```
private void button1_Click(object sender, EventArgs e)
{
    int i;
    MessageBox.Show(string.Format("变量 i={0}", i), "变量初始值");
}
```

(局部变量) int i

错误:
  使用了未赋值的局部变量“i”

图 1-6

要想得到如图 1-3 所示的对话框，在 C#中必须将变量 *i* 进行初始化，给变量赋值为 0，修改后的代码如下：

```
private void button1 Click(object sender, EventArgs e)
{
    int i=0;
    MessageBox.Show(string.Format("变量 i={0}", i), "变量初始值");
}
```

而在 Dev-CPP 环境中编写以下 C 语言程序：

```
int main()
{
    int i;
    printf("变量 i=%d",i);
    getch();
    return 0;
}
```

编译时不会提示错误，运行时则将显示类似图 1-7 所示的结果。

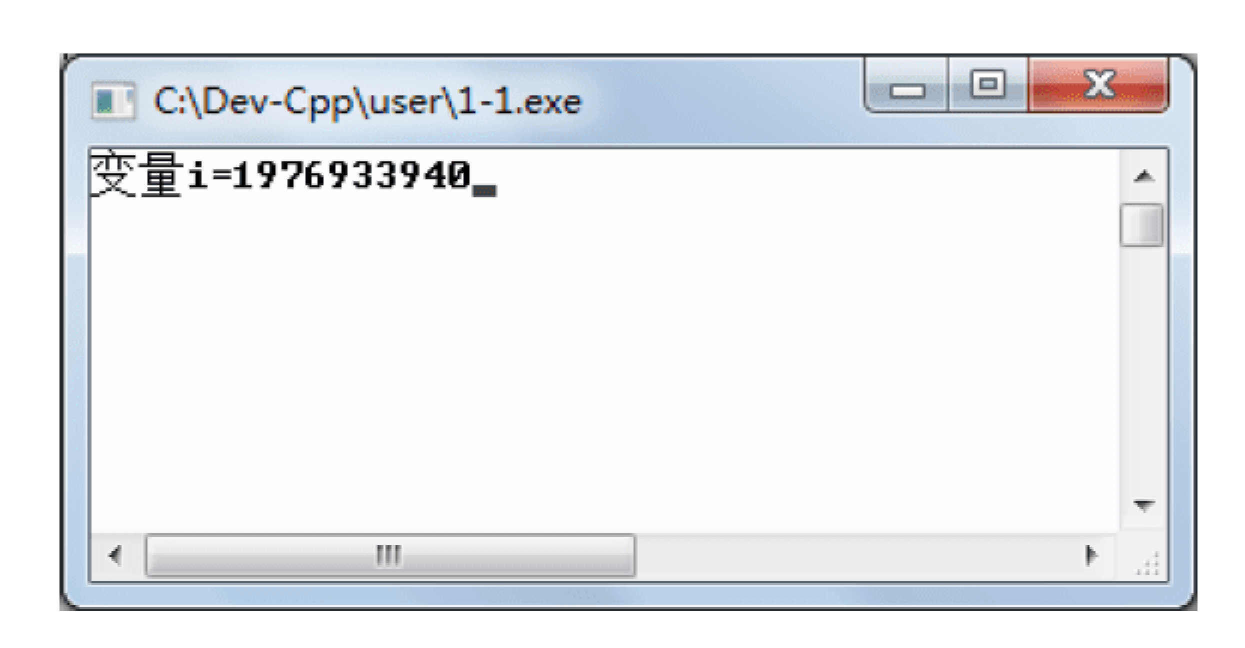


图 1-7

虽然在程序中没有初始化变量 *i*，但变量 *i* 却有一个值（图 1-7 中显示的是 1976933940，下次运行该程序时可能又是另一个值），这是为什么呢？原来，在 ANSI C 中定义变量时，编译器将给该变量分配内存，但并不会将分配的内存初始化为 0。这样，原来该内存区域中保存的是什么值，新指定的变量也就具有了什么值。在图 1-7 所示结果中，给变量 *i* 分配的内存中的值正好为 1976933940，所以变量 *i* 也就具有了这个值。

2．数值0的类型转换

程序中经常会用到数据类型的转换，如将数值类型转换为字符串类型、将数值类型转换为布尔类型等。

将数值 0 转换为字符串 0，这种转换很好理解，其显示的内容都是相同的 0，只有在进行数值运算时才能体现出不同。

数值 0 转换为布尔类型是什么值呢？

在 ANSI C 中没有专门设置布尔类型，在进行逻辑运算时，将 0 值作为布尔值 False，将非 0 值作为布尔值 True。

在 C#中，定义了 Boolean 类型，数值 0 转换为 Boolean 类型时得到的结果为 False，非 0 值转换为 Boolean 类型时得到的结果为 True。

3．除以0异常

我们在小学就学过：0 可以做被除数，但不可以做除数。在程序中，当除数为 0 时，将出现异常。例如，有以下 C 代码：

```
int main()
{
    int Dividend, Divisor,Result;
    Dividend = 8;
    Divisor = 0;
    Result = Dividend / Divisor;
    printf("%d/%d=%d",Dividend,Divisor,Result );
    getch();
    return 0;
}
```

当执行以上代码时，由于除数 Divisor 为 0，将产生一个严重的错误，导致程序不能继续运行，如图 1-8 所示。



图 1-8

在程序执行中如果遇到这种异常，将导致程序中断，但这不是我们所希望的。一个好的程序员应该考虑并处理程序中可能发生的各种异常，并捕获这些异常，然后给用户显示出一个友好的错误提示信息。不过，ANSI C 中并没有提供异常捕获机制，因此需要

程序员根据程序执行过程，主动去判断除数，以避免产生这种严重异常。例如，可将以上代码修改为以下形式：

```
int main()
{
    int Dividend, Divisor,Result;
    Dividend = 8;
    Divisor = 0;
    if(Divisor==0){
        printf("除数不能为 0! ");
    }else{
        Result = Dividend / Divisor;
        printf("%d/%d=%d",Dividend,Divisor,Result );
    }
    getch();
    return 0;
}
```

编译执行以上程序，将得到如图 1-9 所示的结果，提示了“除数不能为 0!”，程序并没有进入严重异常状态。

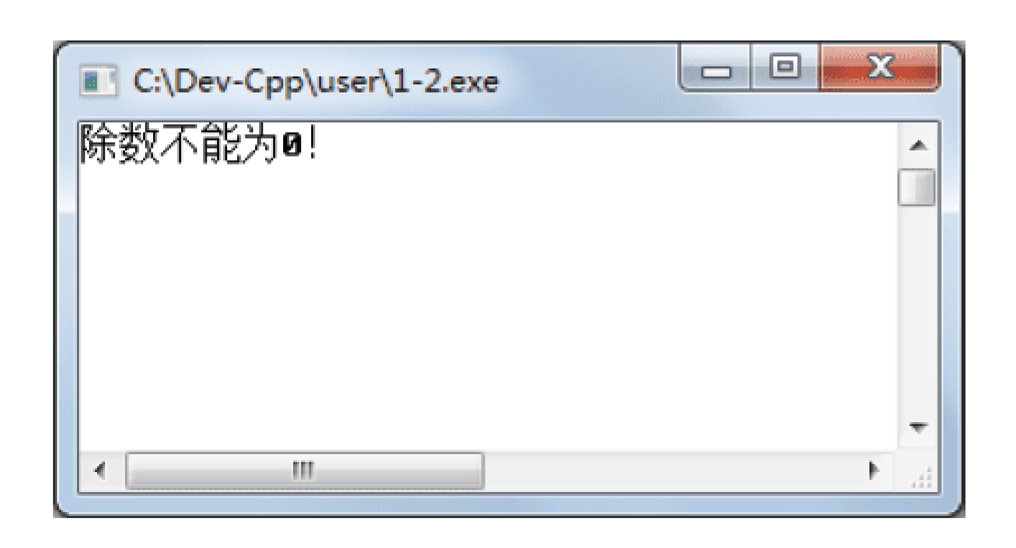


图 1-9

在异常捕获方面，C++、C#就要方便得多。例如，C#定义了很多异常（也包括 DivideByZeroException 异常），我们在程序中可以使用 try…catch 结构来捕获这些异常并进行处理。

1.2　司空见惯的十进制数

有没有想过，为什么 6+8=14？

6+8=14?

从小就这样学的呗！

对，我们小学就开始学“逢十进一，借一当十”，觉得很自然。这就是司空见惯的十进制计数法。

1.2.1　远古的结绳记事

远古时期，人类文明还没有得到发展，但是，“数学”却先于语言、文字而产生。这是因为人们在生活中用到数学的地方很多。例如，每个人捕获猎物的数量，应该怎么表示呢？首先想到的是双手 10 个手指。天长日久，人类在大自然的生存过程中，积累了更多的生存经验。随着人类征服自然、适应自然的能力逐步提高，捕获或养殖的动物数量也逐步增加，此时靠双手的十个手指来计数就不够了。从史料来看，此时人类进一步的做法是排石子、划道道等。此时，数数还不会，计算更是谈不上。

在我国民间有一种传说，认为伏羲氏始创了结绳记事的方法。结绳记事，是在绳子上打一个结来表示一个数，如图 1-10 所示，“事大大其绳，事小小其绳，结之多少，随物众寡”，这在当时所起的作用是非常大的。

随着人类文明的进步，人类将一只羊、两只羊、三只羊……这些具体的概念抽象化，得到了数字 1、2、3……，只是当时的表现方式有所不同，如图 1-11 所示。从图中可看到，巴比伦数字类似于按数量摆放石子，而中国数字类似于画痕，罗马数字进一步抽象，用Ⅴ表示数字 5，如果在其左侧有一竖，表示为 4（=5−1），若在其右侧有一竖，表示为 6（=5+1），右侧有两竖，表示为 7（=5+2），依次类推。在罗马数字中用Ⅹ表示 10，根据其左侧或右侧的竖线数量来表示低于 10 或大于 10 的数。现在罗马数字仍在很多地方使用。



图 1-10

图 1-11

阿拉伯数字则是现今国际通用的数字，最初由印度人发明，后由阿拉伯人传向欧洲，之后再经欧洲人将其现代化。正因阿拉伯人的传播，成为该种数字最终被国际通用的关键点，所以人们称其为“阿拉伯数字”。阿拉伯数字由 0、1、2、3、4、5、6、7、8、9 共 10 个计数符号组成。采取位值法，高位在左，低位在右，从左往右书写。借助一些简单的数学符号（小数点、负号等），这个系统可以明确地表示所有的有理数。为了表示极大或极小的数字，人们在阿拉伯数字的基础上还创造了科学记数法。

1.2.2　什么是十进制计数

正如本节开始时所说，十进制计数法是我们司空见惯的，从小学习的就是十进制。那么，什么是十进制计数？

十进制数基于位进制和十进位两条原则，即所有的数字都用 10 个基本的数字表示，满 10 进 1，同时同一个数字在不同位置上所表示的数值大小不同，因此数字的位置非常重要。

十进制的基本数字是 0、1、2、3、4、5、6、7、8、9。要表示这 10 个数字的 10 倍，就将这些数字左移一位，右侧用 0 补上空位，即可得到 10、20、30、…90（0 的 10 倍还是 0），如图 1-12 所示。若要继续扩大 10 倍来表示数字，就继续左移数字的位置，然后在右侧用 0 补上空位，即 100、200、300…。

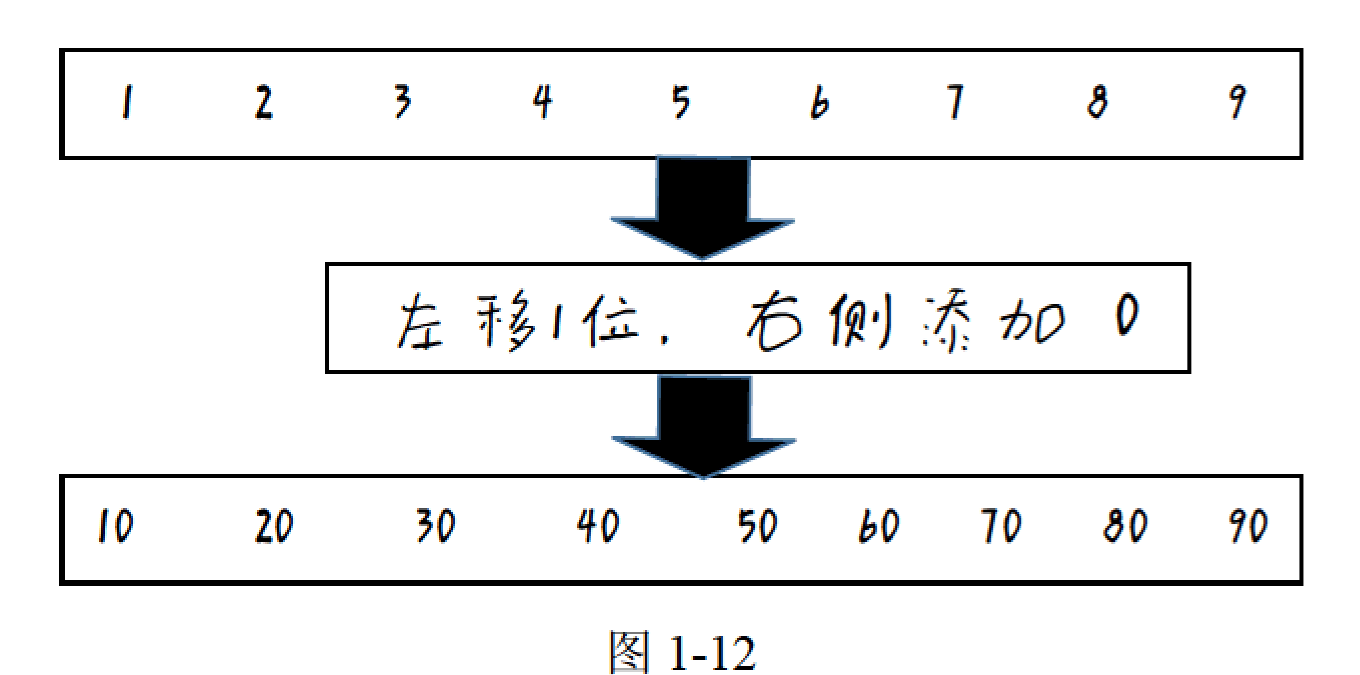


图 1-12

要表示一个数的十分之一，百分之一，千分之一，就将数字向右移，在左侧（小数点右侧）补上 0，即可得到十分位（0.1）、百分位（0.01）、千分位（0.001），如图 1-13 所示。

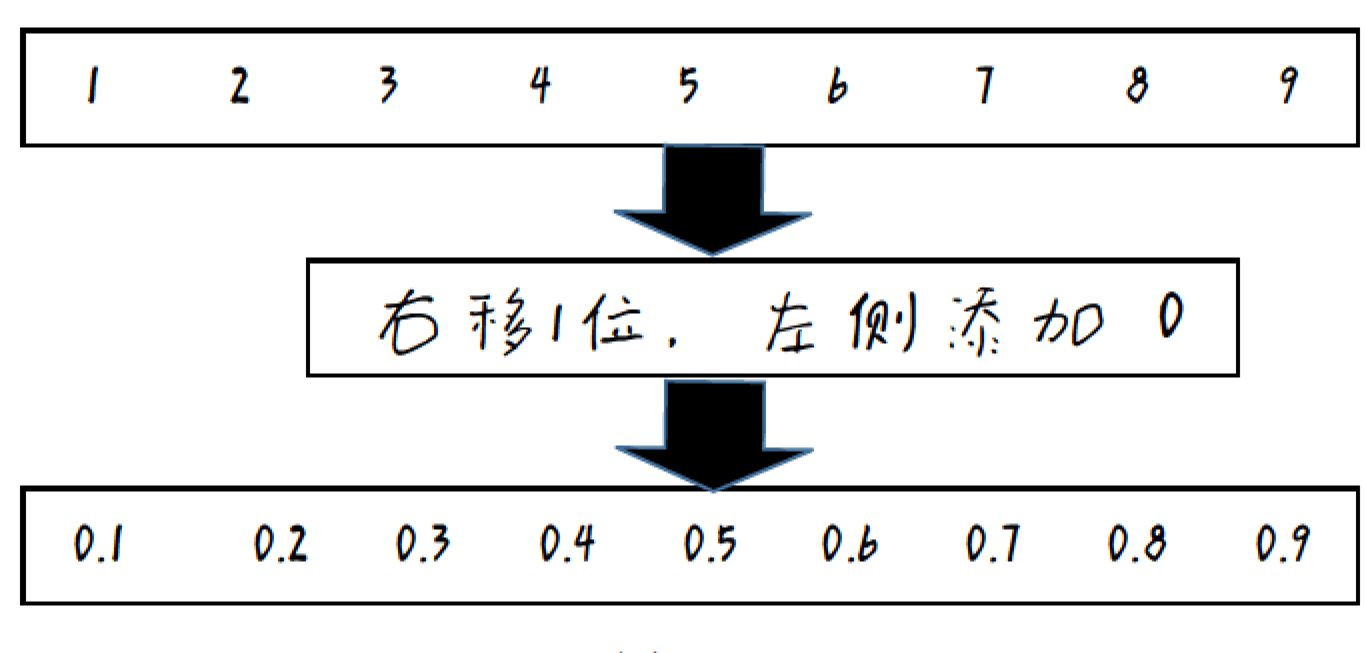


图 1-13

1.2.3　为啥人类习惯十进制

为什么我们从小学习的就是十进制，而不是更简单的二进制？

首先，看看我们的双手，我们有 10 根手指，如图 1-14 所示。从人类最初计数时起，首先想到的就是用双手的手指来计数，数满 10 个数再增加一双手，这样就产生了十进制。



图 1-14

另一个很重要的方面，就是习惯。我们从小接受的教育就是使用十进制数进行计算，因此习惯了十进制数的运算。

二进制的运算规则比十进制简单，为什么不使用二进制呢？这是因为二进制的运算规则虽然简单，但是要表示一个较大的数据时，需要用很长一串数据，如十进制的 50000 写成二进制为 1100 0011 0101 0000，一共需要 16 位，谁一眼能看出该数据的大小？

$$(50000)_{10}=(1100\ 0011\ 0101\ 0000)_2$$

如果我们一直使用二进制，可能对二进制表示的数也能方便地识别出来，但是和十进制相比可以看出，十进制数比二进制数更简洁，更易识别。

而比十进制更大的进制（如十六进制），其运算规则复杂，更难以使用。

因此，在日常生活中是以十进制数为主。

1.2.4　十进制运算规则

十进制数的常用运算包括加、减、乘、除这 4 种，也称为四则运算。在初等数学中，当一级运算（加、减）和二级运算（乘、除）同时出现在一个算式中时，它们的运算顺序是先乘除，后加减。要改变这种运算规则，则需要通过括号，因为四则混合运算中，总是先计算括号内，然后再计算括号外。同一级运算顺序则是按从左到右的顺序进行。

在加、减、乘、除这 4 种运算中，加、减法互为逆运算，乘、除法互为逆运算，而

乘法是加法的简便运算，如图 1-15 所示。

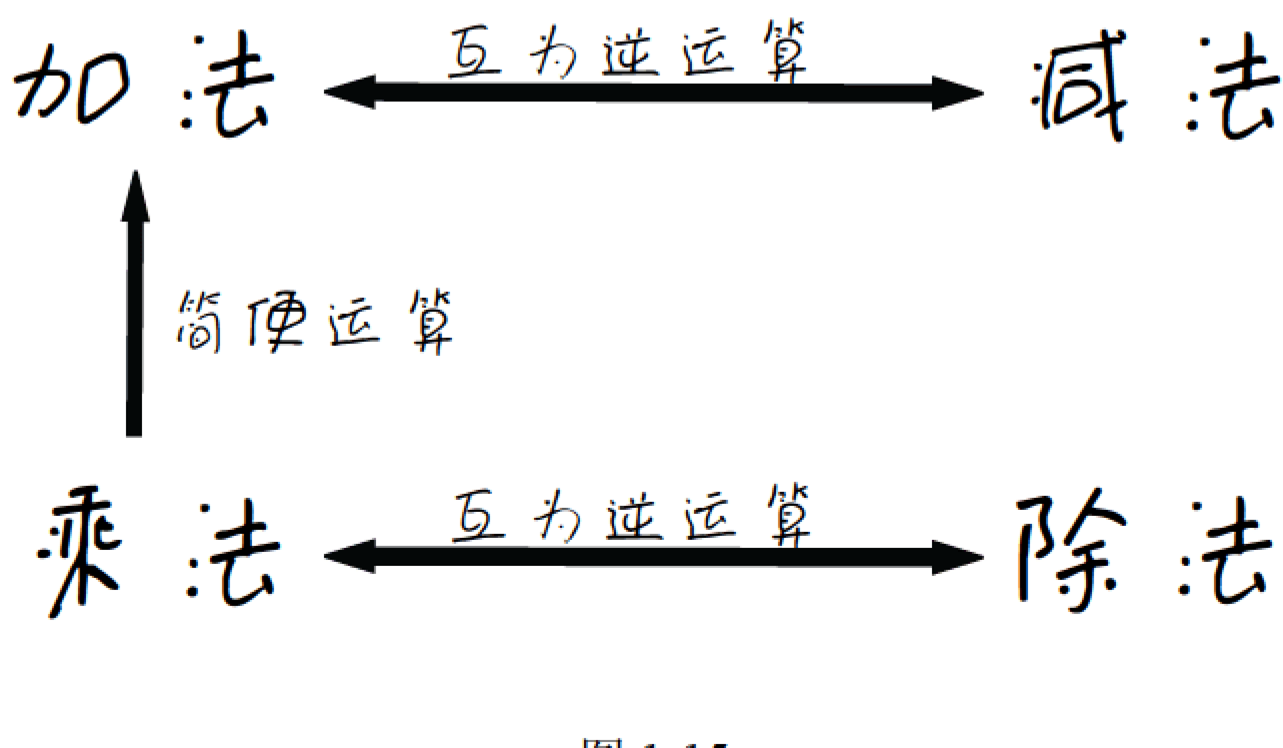


图 1-15

1. 加法

加法运算是把两个数合并成一个数的运算，可以将整数、小数、分数进行合并运算。在加法运算中，首先应当将相加的数从个位开始按位对齐，然后从个位开始（从右向左）逐位相加。加法运算时，两数（或多数）相加的和超过 10 时就向前一位进位，这种规则称为“逢 10 进 1”，如图 1-16 所示。

2. 减法

减法运算是已知两个加数的和与其中一个加数，求另一个加数的运算。在减法运算中，首先应当将相减的数从个位开始按位对齐，然后从个位开始逐位相减。如果对应位上被减数小于减数时，需向被减数前一位进行借位，借 1 当 10，再和本位的数相加，得到一个超过 10 的数，再用这个数与减数进行运算即可，如图 1-17 所示。

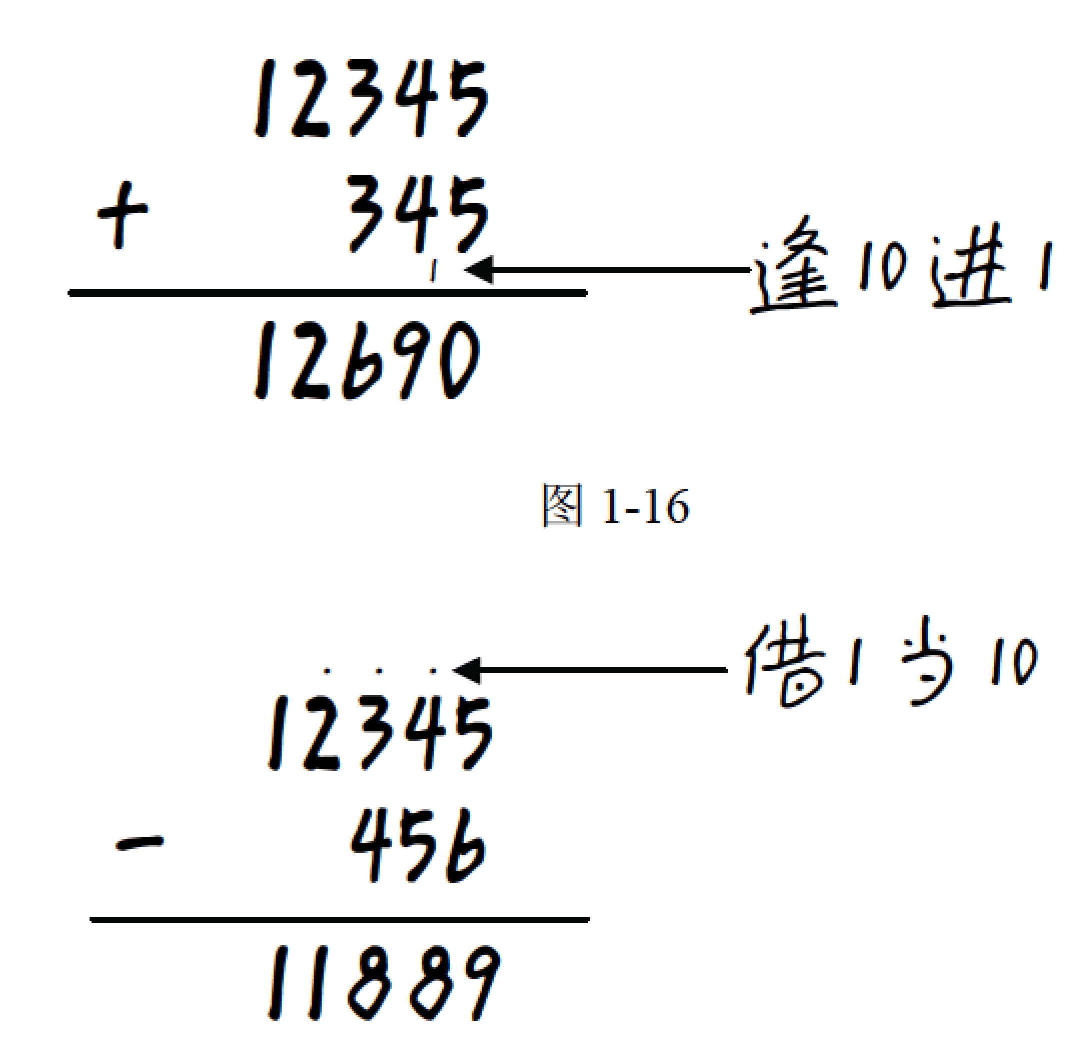


图 1-16

图 1-17

3．乘法

乘法运算是求几个相同加数的和的简便运算。乘法运算比加、减法更复杂，不过，对于十进制数的乘法运算来说，只需要背熟九九乘法表，并按此规则逐位相乘，然后再将各位乘积进行累加，即可得到最终结果，如图 1-18 所示。

```
     12345
×       23
----------
     37035
+   24690
----------
    283935
```

图 1-18

4．除法

除法运算是已知两个因数的积与其中一个因数，求另一个因数的运算。

除法法则：除数是几位，先看被除数的前几位，前几位不够除，多看一位，除到哪位，商就写在哪位上面，不够商 1 时，要用 0 占位。除法可能会有余数，余数要比除数小。如果商是小数，商的小数点要和被除数的小数点对齐；如果除数是小数，要将其化成整数后再用整数的除法进行计算。

除法是乘法的逆运算。图 1-18 所示的乘法算式，可表示成如图 1-19 所示的除法算式：

$$283935 \div 23=12345$$
$$12345 \times 23=283935$$

图 1-19

1.2.5 十进制数的分解

十进制数由 0～9 这 10 个数字组成，依据数字所在位置决定数值的大小。数据的各位从右向左依次为个位、十位、百位、千位……。

如图 1-20 所示，个位的 9 表示有 9 个 1，十位的 8 表示 8 个 10，百位的 7 表示 7 个 100，按这种方式，可将十进制数按图 1-21 所示方式进行分解。

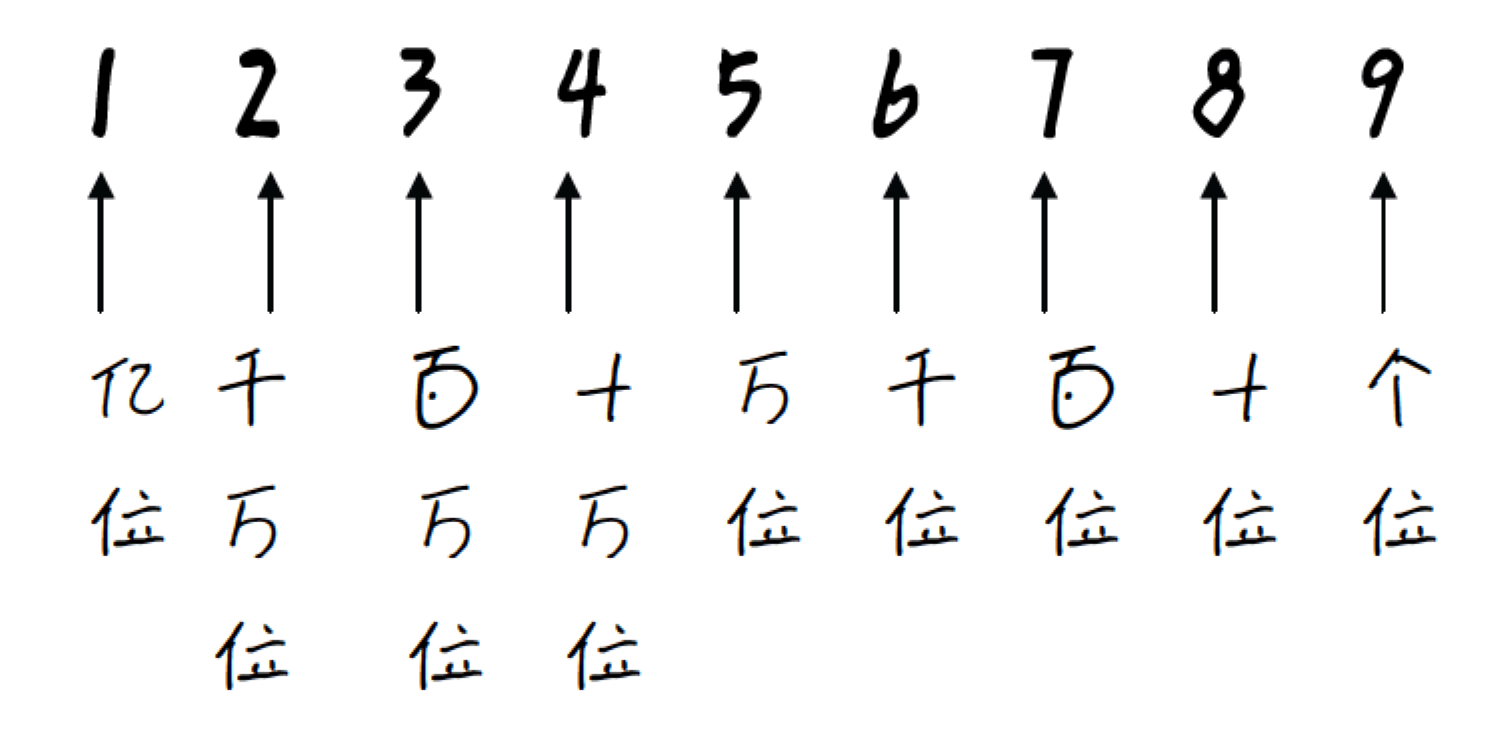


图 1-20

$$
\begin{aligned}
&123456789\\
&=1\times 100000000+2\times 10000000+\cdots+8\times 10+9
\end{aligned}
$$

图 1-21

仔细看图 1-21 所示数据的分解式，从右向左，每个数码都比其右侧的数大 10 倍，可将上式简写成图 1-22 所示的形式。

$$
\begin{aligned}
&123456789\\
&=1\times 10^8+2\times 10^7+\cdots+8\times 10^1+9\times 10^0
\end{aligned}
$$

图 1-22

十进制是我们从小就开始学的，以上这些规则都很简单，为什么还要在这里重复呢？因为程序员通过十进制的这些运算规则可推导出其他进制数的运算规则，也可设计解决更多的问题，例如大整数的运算问题。

1.2.6　20！等于多少

在设计大整数之前，我们先来看一个例子。以下是一个 C 语言程序，该程序中定义了一个计算整数阶乘的函数 fact()，在主函数中调用 fact()函数计算 1～20 各数的阶乘。

```
int fact(int f)
{
    int result=1,i;
    for(i=2;i<=f;i++)
    {
      result*=i;
```

```
    }
    return result;
}

int main()
{
     int f,r;
     for (f=1;f<=20;f++)
     {
          r=fact(f);
          printf("%d!=%d\n",f,r);
    }
    getch();
    return 0;
}
```

运行以上程序，得到如图 1-23 所示的结果。

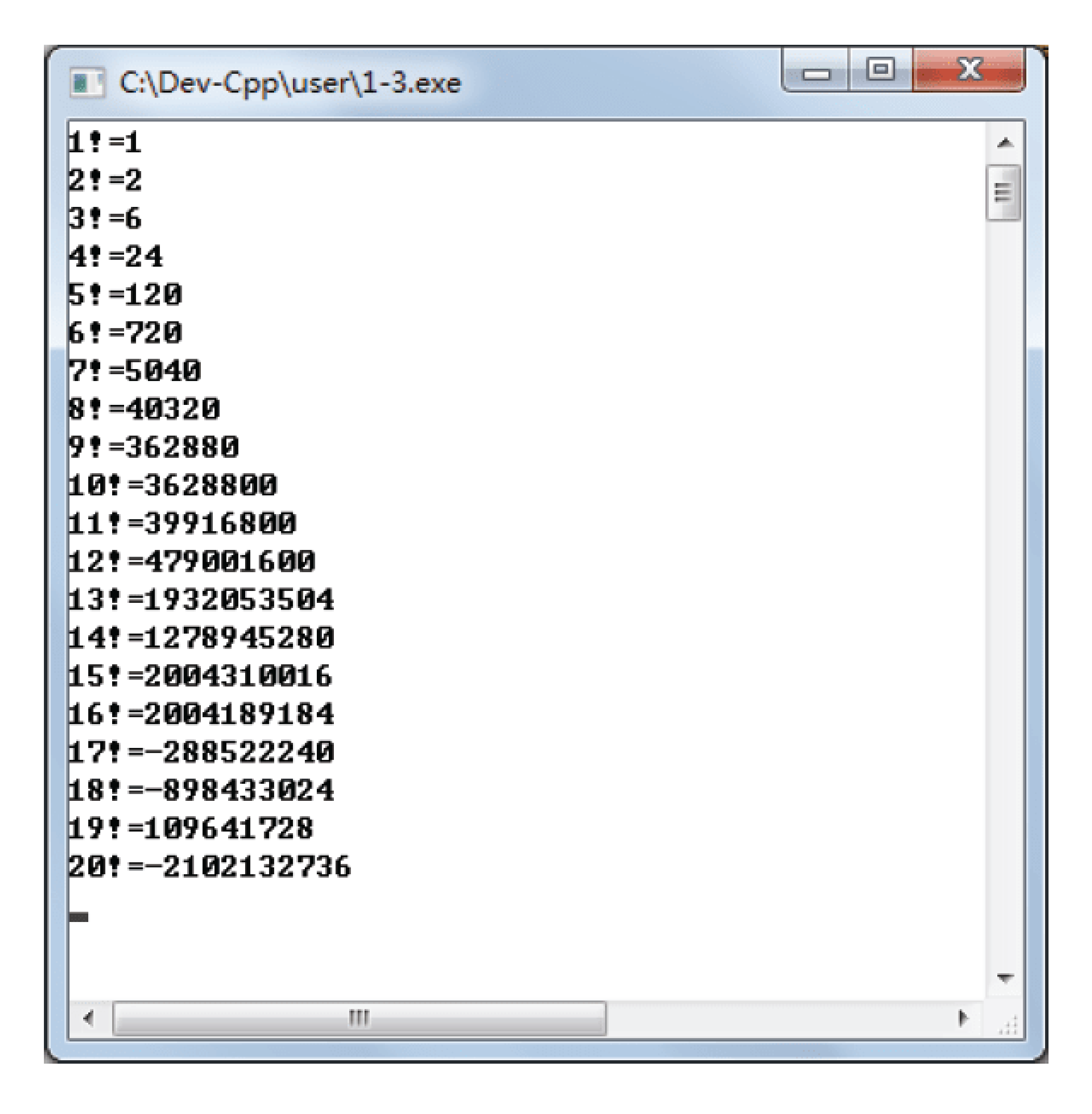


图 1-23

从图 1-23 中可以看出，14!的结果比 13!的结果还小，肯定出问题了。为什么会出现这种错误呢？电脑连 15 的阶乘都计算不出来？

分析程序代码可看出，程序中使用的是 int 类型的变量，在 C 语言中，这种变量保存的数据范围为–2,147,483,648～2,147,483,647，而 13!再乘以 14，其结果已经超过 int 类型的表示范围（其实，13!的值应该为 6,227,020,800，已经超过了 int 类型的表示范围），因此，数据就出错了，从 17!、18!还可以看出其结果变成了负数。

既然知道了出错原因是由于数据类型导致的，那么，是不是将以上程序的 int 类型

改变为位 long 类型，就可以计算更大数的阶乘了呢？理论上是这样，不过，由于 ANSI C 中规定，在字长为 32 位的计算机中，int 类型和 long 类型都是 32 位。因此，这里将数据类型修改为 long 也不能解决问题。

在支持 64 位字长的 C#系统中，long 类型使用 64 位二进制位表示（8 个字节），其表示的数据范围为–9,223,372,036,854,775,808～9,223,372,036,854,775,807。但是，这么大的数在阶乘面前也很快就被会填满，图 1-24 所示为使用 C#计算各数阶乘的输出结果。

从图 1-24 中可看到，使用 64 位字长的 long 类型，20 的阶乘也可以被正确表示出来了。但是更大的数呢？21!、22!的结果是多少？

```
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5,040
8!=40,320
9!=362,880
10!=3,628,800
11!=39,916,800
12!=479,001,600
13!=6,227,020,800
14!=87,178,291,200
15!=1,307,674,368,000
16!=20,922,789,888,000
17!=355,687,428,096,000
18!=6,402,373,705,728,000
19!=121,645,100,408,832,000
20!=2,432,902,008,176,640,000
21!=-4,249,290,049,419,214,848
22!=-1,250,660,718,674,968,576
```

图 1-24

哦，My God！还是出错了，21 的阶乘就变成负数了。这还是 long 类型的表示范围问题，long 类型的表示范围为–9,223,372,036,854,775,808～9,223,372,036,854,775,807。

对于基本的整数类型，使用 ulong（无符号长整型）类型来保存数据，其表示范围也只为 0～18,446,744,073,709,551,615，再大的数就没办法表示了。那么，更大数的阶乘该怎么办呢？

1.2.7 大整数构想

在实际应用中，除了阶乘之外，还有很多地方需要使用到非常大的整数，而计算机

程序设计语言对数据的表示范围总是有限的。因此，还得我们程序员自己想办法，设计一个能处理大整数的类，这个类应该能处理任意位数长度的整数。

根据本节前面对十进制数的分析，可以很容易地想到，可以在程序中用一个数组来表示大整数的各位，由于数组元素的多少只受计算机内存限制，因此，就可以处理任意长度的大整数了。

如图 1-25 所示，定义一个数组，然后将数据的各位分解到数组的各个元素中。

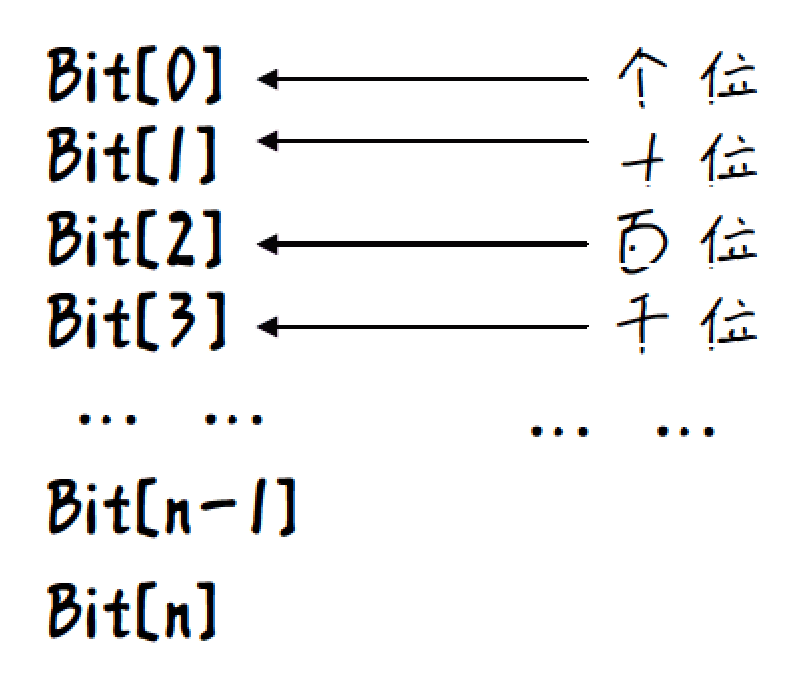


图 1-25

这个数组只是大整数的一种表示形式，要处理大整数，还需要记录数据的正负、加减时的进位等。然后定义以下常用操作：

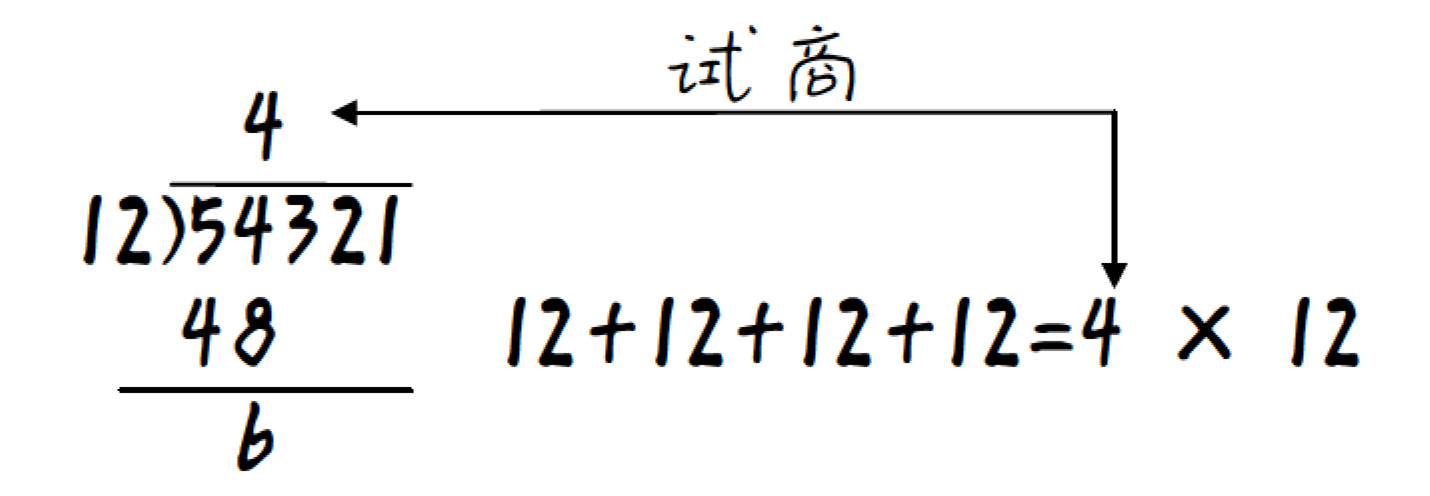
- 加法：将整数从个位开始（即数组的 0 号元素）进行累加。累加时还需判断是否要进位（逢 10 进 1），因此，在累加时还需要将进位数进行累加。
- 减法：将整数从个位开始（即数组的 0 号元素）进行减法操作，若不够减还需要从前一位借位（借 1 当 10）。被减数在进行减法操作时还需要减去被借的位。
- 乘法：按图 1-18 所列的乘法算式，可将乘数中的每一个数组元素与被乘数相乘，然后将结果累加，即可得到大整数相乘的结果。当然，在进行加法和乘法运算时也需要考虑进位的情况。
- 除法：除法的实现要麻烦一点。首先要考虑试商的问题，从被除数的高位开始与除数对齐，试商时用被除数的部分位减去除数，判断能减几次，就可商几。另外，除法还需要考虑余数问题。除法的过程如图 1-26 所示。

从图 1-26 的演算过程可看到，除法运算需要循环调用加法运算进行试算，然后再调用减法运算计算试商后的余数。

根据这个构想编写大整数处理函数，即可处理任意长度的整数（如可保存、计算长度为 100 位、200 位甚至更多位整数的加、减、乘、除运算），不再局限于 C 语言所提供数据类型中有限的整数长度了。

有幸的是，在微软的.NET Framework 4（以及 JAVA 的 JDK 1.5）中已经提供了一个大整数类型，可以处理任意长度的大整数。如果使用.NET Framework 4 进行开发，就不

用自己编写大整数类型了。当然，如果在 ANSI C 环境下编写程序，仍然可以按本节介绍的构思编写自己的大整数类型。



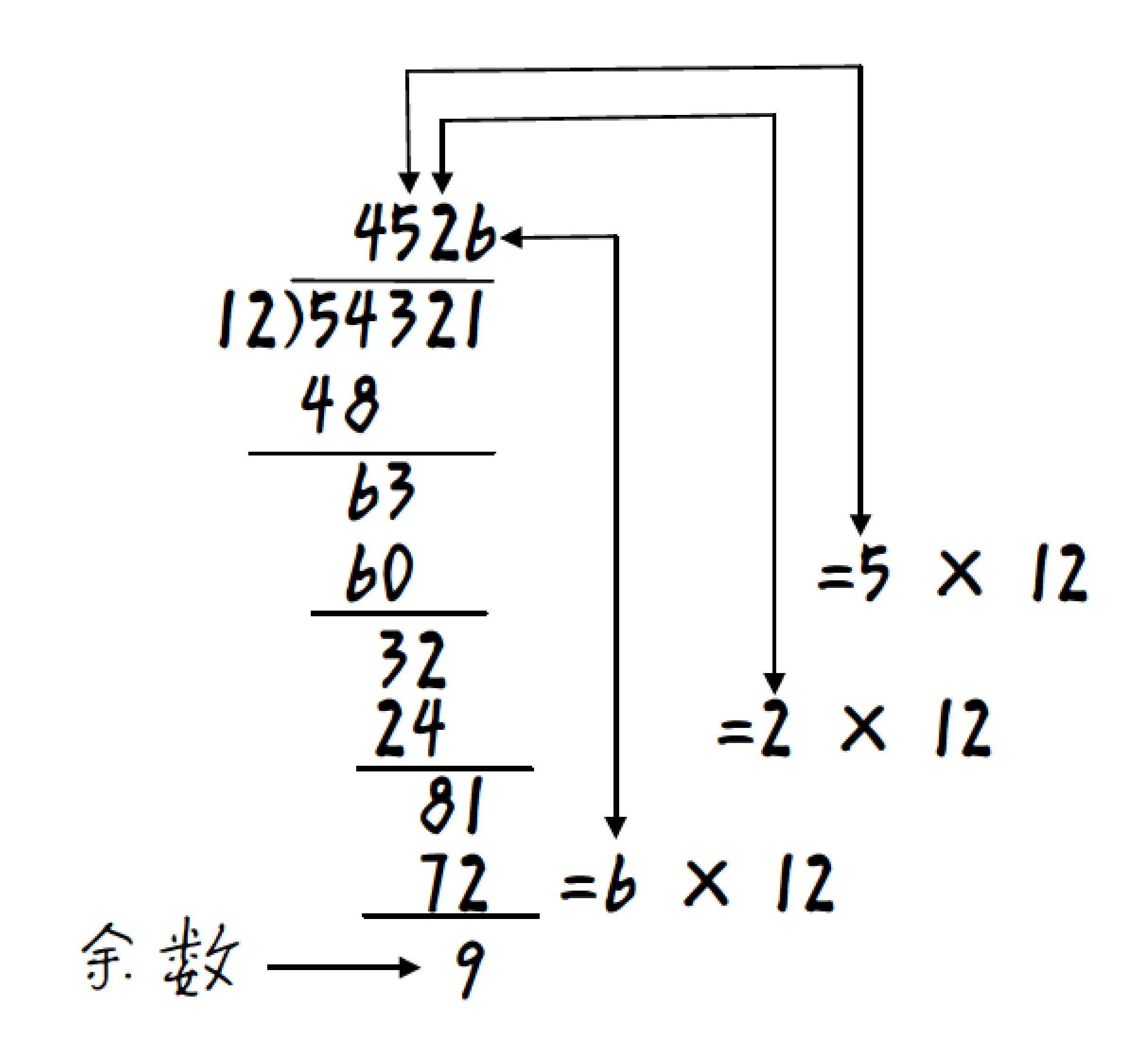


图 1-26

1.3　为啥要用二进制

既然人类从远古时代就开始使用十进制数了，为啥还要用二进制呢？二进制有什么优势呢？下面我们一起走进二进制的世界。

1.3.1　人脑与电脑

通过人类的进化，以及人们长期的学习和训练，人脑的潜能可以被不断地发掘。吉尼斯世界纪录中记纸牌记得最多的是一名英国人，他只需看一眼就能记住 54 副洗过的扑克牌（一共有 2916 张牌）。还有人能记住圆周率小数点后的 42,905 位数字！可见，人脑的潜能是可以不断被挖掘的。

在生活中，人脑对很多事物都形成了条件反射，例如，对于数据 10 与 9 的大小，我们可以直接反应出 10 比 9 大。不过，由于没有通过相应的运算，仅凭人脑直觉反应得出的结果可能是不准确的。例如，对于像比较大的两个数 99999999 与 100000000，要想看一眼就得出哪一个数据更大，就变得不太可能了，即使得出结论，可能也会有错误。为什么呢？这是因为数据的位数变多了，并且重复的数很多，人脑无法一下子反应出来。通常要数一下有多少位数，然后才能进行判断。

计算机（俗称的电脑）却不一样，对于任何操作，电脑都需要经过相应的运算，然后才能得出结果。不管是比较 10 与 9，还是比较 99999999 与 100000000，电脑都会按规定的算法进行运算，最后得出相应的结果。而电脑一旦得出结果，其结果肯定是准确的！

在电脑中，使用二进制来保存数据和编写程序。为什么选用二进制，而不选用人类已经熟练使用的十进制呢？

如果要让电脑使用十进制，首先，应该让电脑能识别出十进制中的 10 个数字。怎么识别 10 个数字呢？通常的考虑是，可以通过元器件中电压的高低水平来分别标识 10 个数字。假如最高电压为 12V，那么 10 个数字中，每个数码可以分配的电压区间为 1.33V，如图 1-27 所示。

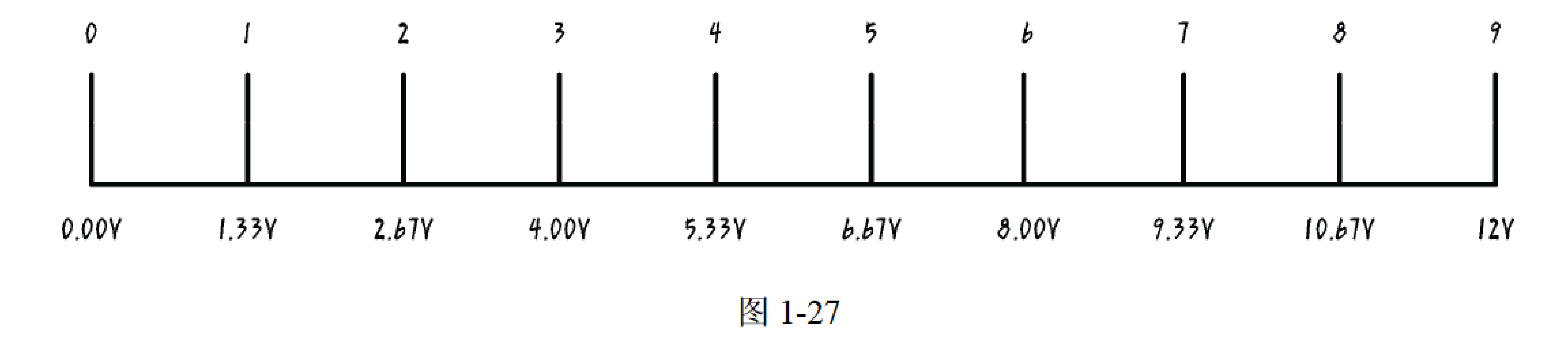


图 1-27

从图 1-27 可知，每个数之间的电压间隔小，如果外界干扰造成电压大幅变化，数据就不准确了（如本来电压为 1.33V，可被识别为数字 1，但是由于外界干扰，电压增加了 1V，就变成 2.33V 了，这里距离 2.67V 更近，就可能被识别为数字 2）。还有一个最大的问题，在硬件上要识别这 10 种状态，其电路结构将非常复杂。

当然，这里只是一种假设，实际应用中采用的是二进制。由于二进制数只有 2 个数码，电路就很简单了，因为具有两种稳定状态的元件（如晶体管的导通和截止，继电器的接通和断开，电脉冲电平的高低等）很容易被找到。

因此，在电脑中使用二进制主要有以下优点：

- 技术实现简单。电脑由逻辑电路组成，逻辑电路通常只有两个状态，开关的接通与断开，这两种状态正好可以用“1”和“0”表示。
- 运算规则简单。两个二进制数的和、积运算组合分别有 3 种规则，相比十进制数的运算规则来说非常简单（十进制的九九乘法表就有 81 种规则），有利于简化计算机内部结构，提高运算速度。

- ❑ 适合逻辑运算。逻辑代数是逻辑运算的理论依据，二进制只有两个数码，正好与逻辑代数中的“真”和“假”相吻合。
- ❑ 易于进行转换。二进制数与十进制数、八进制数、十六进制数之间的转换很方便。
- ❑ 抗干扰能力强。用二进制表示数据具有抗干扰能力强、可靠性高等优点。因为每位数据只有高低两个状态，当受到一定程度的干扰时，仍能可靠地分辨出它是高电平还是低电平（只分辨电平的高低，而不用识别具体电压值）。

知道电脑用二进制数来存储后，再来看电脑就简单了。电脑保存的数据用电路的两种状态表示，当数据很大时，只需要增加数据的位数就可以了。

当超大规模集成电路迅速发展起来后，电脑中就可以处理、存储海量数据信息了。并且，由电脑保存的数据保存周期长，不易丢失、损坏。而由于人类的认知及人的记忆会随时间出现遗忘等原因，要用人脑来存储、处理海量信息，就不太可能。

因此，很多人认为电脑比人脑强。其实，电脑本质上只能识别 0 和 1 这两种状态！而更复杂的功能，则是由人类对 0 和 1 这两种状态进行各种组合而得到的。

1.3.2 二进制计数规则

二进制的计数规则非常简单，只需要记住以下 3 点就行了：

- ❑ 基数为 2。
- ❑ 只有 2 个数码，即 0 和 1。
- ❑ 逢 2 进 1，借 1 当 2。

如图 1-20 所示，十进制数可以由多位组成，从右向左分别为个位、十位、百位、千位、万位……，与此类似，二进制数也可由多位组成，从右向左分别为 1 位、2 位、4 位、8 位、16 位……，如图 1-28 所示。

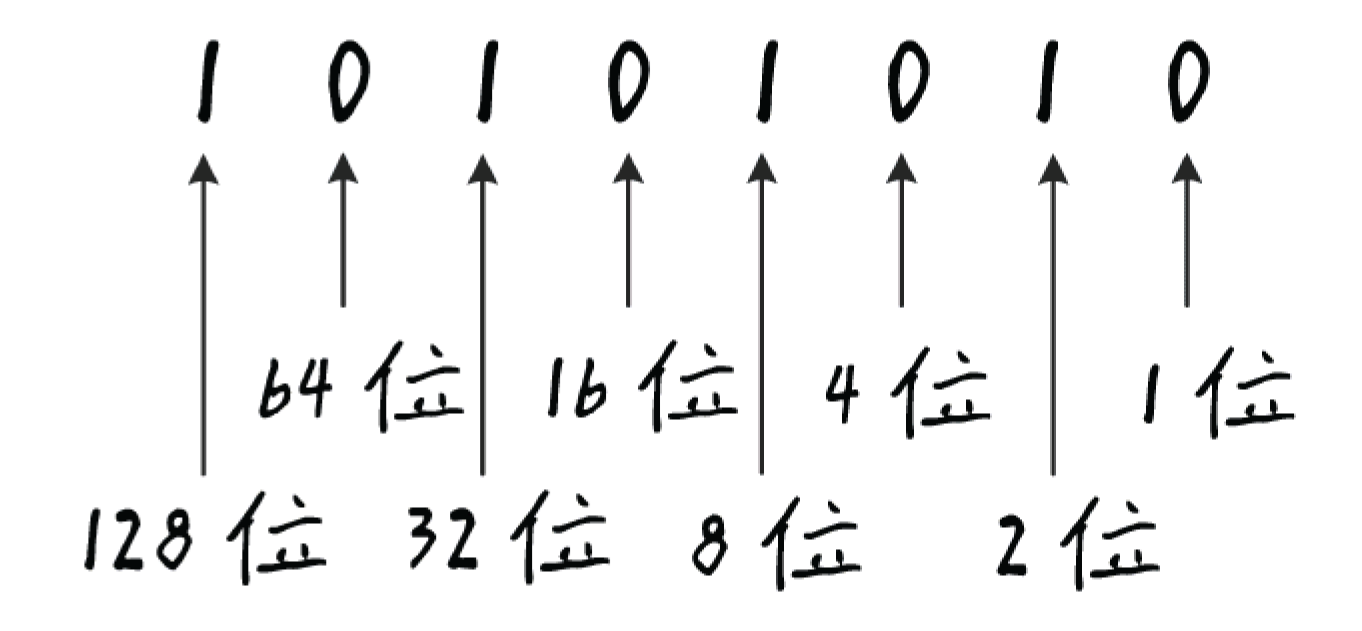


图 1-28

为什么称为 1 位、2 位、4 位、8 位……呢？其实，这是从十进制角度来看二进制的各位数得出的名称。根据二进制计数的规则，用二进制计数时，第一个数为 0，第 2 个

数为 1（根据逢 2 进 1 的规则），接下来为 10（所以第 2 位就是“2 位”），继续下来依次为 11、100（所以第 3 位就是 4 位）、101、110、111、1000……。

如表 1-1 所示是十进制数 0～127 用二进制表示的对应形式。从表 1-1 中可以看到，当十进制数为 0 和 1 这两种情况时，可用 1 位二进制数来表示这两种状态；当十进制数在 7 以内时，可以使用 3 位二进制数来表示——随着十进制数的增大，对应的二进制数的位数也会增多，在表 1-1 中表示到十进制数 127 时就需要 7 位二进制数来表示了。

表 1-1　十进制数 0～127 对应的二进制表示

十进制	二进制	十进制	二进制	十进制	二进制	十进制	二进制
0	0	32	10 0000	64	100 0000	96	110 0000
1	1	33	10 0001	65	100 0001	97	110 0001
2	10	34	10 0010	66	100 0010	98	110 0010
3	11	35	10 0011	67	100 0011	99	110 0011
4	100	36	10 0100	68	100 0100	100	110 0100
5	101	37	10 0101	69	100 0101	101	110 0101
6	110	38	10 0110	70	100 0110	102	110 0110
7	111	39	10 0111	71	100 0111	103	110 0111
8	1000	40	10 1000	72	100 1000	104	110 1000
9	1001	41	10 1001	73	100 1001	105	110 1001
10	1010	42	10 1010	74	100 1010	106	110 1010
11	1011	43	10 1011	75	100 1011	107	110 1011
12	1100	44	10 1100	76	100 1100	108	110 1100
13	1101	45	10 1101	77	100 1101	109	110 1101
14	1110	46	10 1110	78	100 1110	110	110 1110
15	1111	47	10 1111	79	100 1111	111	110 1111
16	1 0000	48	11 0000	80	101 0000	112	111 0000
17	1 0001	49	11 0001	81	101 0001	113	111 0001
18	1 0010	50	11 0010	82	101 0010	114	111 0010
19	1 0011	51	11 0011	83	101 0011	115	111 0011
20	1 0100	52	11 0100	84	101 0100	116	111 0100
21	1 0101	53	11 0101	85	101 0101	117	111 0101
22	1 0110	54	11 0110	86	101 0110	118	111 0110
23	1 0111	55	11 0111	87	101 0111	119	111 0111
24	1 1000	56	11 1000	88	101 1000	120	111 1000
25	1 1001	57	11 1001	89	101 1001	121	111 1001
26	1 1010	58	11 1010	90	101 1010	122	111 1010
27	1 1011	59	11 1011	91	101 1011	123	111 1011
28	1 1100	60	11 1100	92	101 1100	124	111 1100
29	1 1101	61	11 1101	93	101 1101	125	111 1101
30	1 1110	62	11 1110	94	101 1110	126	111 1110
31	1 1111	63	11 1111	95	101 1111	127	111 1111

1.3.3 简单的二进制运算规则

与十进制数的运算规则相比，二进制数的运算规则就简单多了。同样，二进制也可对数据进行加、减、乘、除这些基本的算术运算，另外，二进制数据还可进行逻辑运算。有关逻辑运算的内容需要另一个主题来介绍，下面先来看看二进制的算术运算是多么的简单。

1. 加法

与十进制的加法相比，二进制的加法规则要简单得多。如果只考虑一位数相加的情况，在十进制加法运算中，加数和被加数都有 0～9 共 10 种可能，因此，会产生 100 种可能的情况（即使根据加法交换律将重复的运算过滤掉，也会有 55 种情况）。而二进制加法运算中，加数和被加数都只有两种可能，因此，只会有以下 4 种情况：

0+0=0
0+1=1
1+0=1
1+1=10 （逢2进1）

根据加法交换律，将第 2、3 种运算看作为一种运算，则二进制的加法运算就只有 3 种情况了。3 比 55！二进制的加法运算规则是不是要简单得多！

对于多位数的二进制相加，其运算规则与十进制相同，仍然会有进位的情况，只是进位时采用“逢 2 进 1”的方式。例如，将十进制数 26 加上 39，根据表 1-1 找出对应的二进制数，即可列出如图 1-29 所示的十进制数与二进制数加法的竖式。

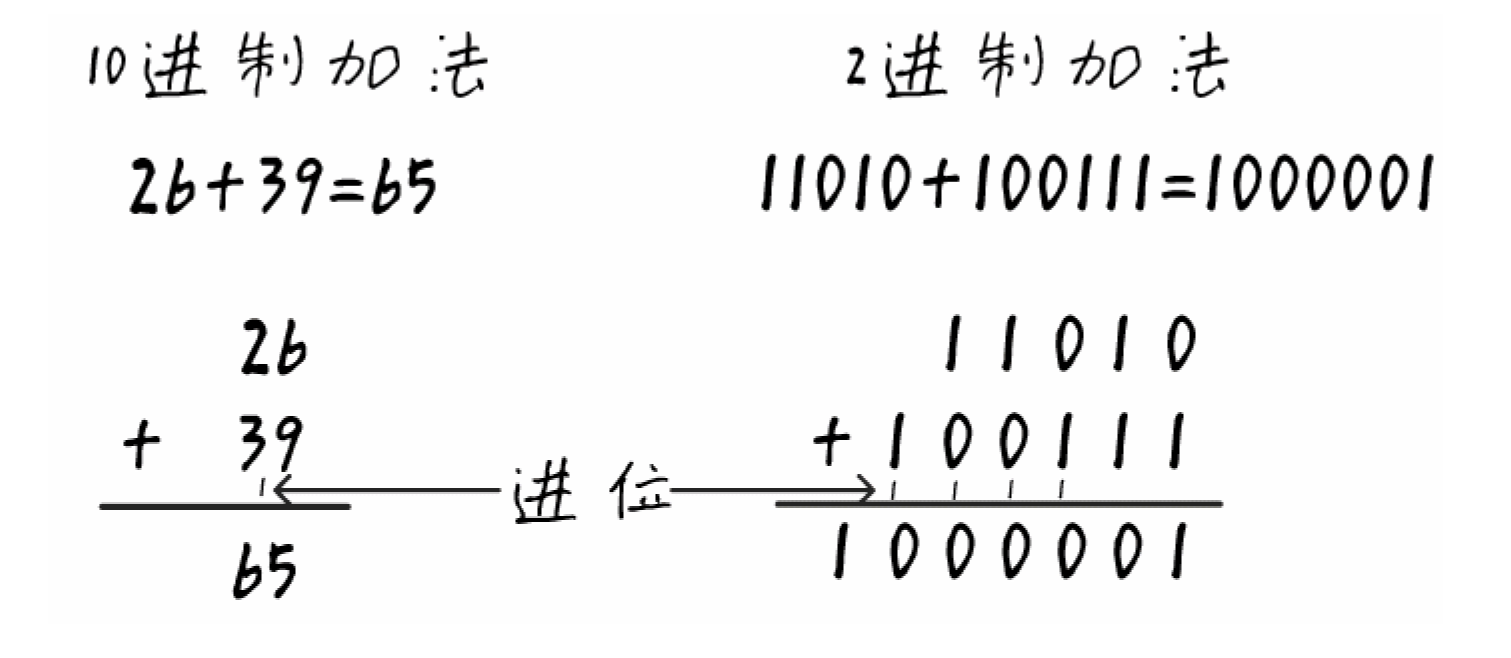


图 1-29

在图 1-29 所示的两种进制的加法运算中，左侧的十进制运算是按“逢 10 进 1”的方式进位，右侧的二进制运算则是按“逢 2 进 1”的方式进位。虽然是两种不同的数据表示方式，但运算的结果是一致的（十进制数 65 对应的二进制数为 1000001）。

2. 减法

二进制的减法运算规则也很简单，只有以下 4 种可能：

0-0=0

1-1=0

1-0=1

0-1=1（借 1 当 2）

如图 1-30 所示为十进制和二进制减法的竖式，在运算时注意十进制是“借 1 当 10”，而二进制则是“借 1 当 2”。

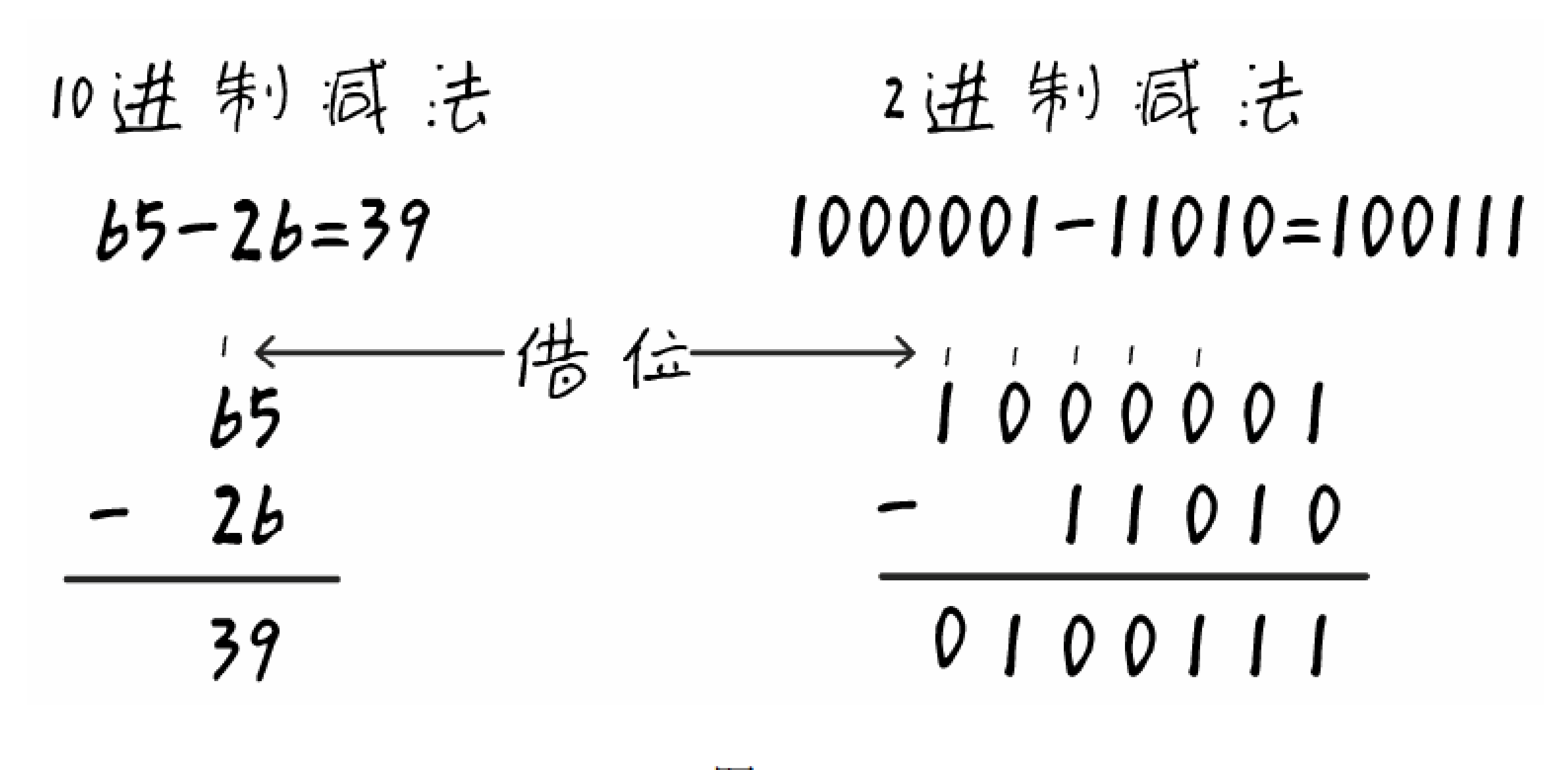


图 1-30

3. 乘法

十进制数的乘法运算需要按“九九乘法表”法则进行，而二进制乘法的规则就简单多了，与加、减法类似，也只有以下 4 种情况：

0 × 0=0

0 × 1=0

1 × 0=0

1 × 1=1

可以看出，只有当被乘数、乘数都为 1 时，结果才为 1；当被乘数或乘数有一个为 0 时，相乘的结果就为 0。

另外，二进制的乘法运算可以很简单地转化为加法运算。首先看如图 1-31 所示的乘法竖式，左边是十进制的乘法竖式，右边是二进制乘法竖式。

从右侧的乘法竖式可看出，当乘数某位为 0 时，这一位与被乘数相乘时各位都为 0，当乘数某位为 1 时，这一位与被乘数相乘时得到的是被乘数对应的各位，只是需要将被

乘数向左移动相应的位数。这样，二进制数的乘法就可以很简单地转化为“加法与移位”。

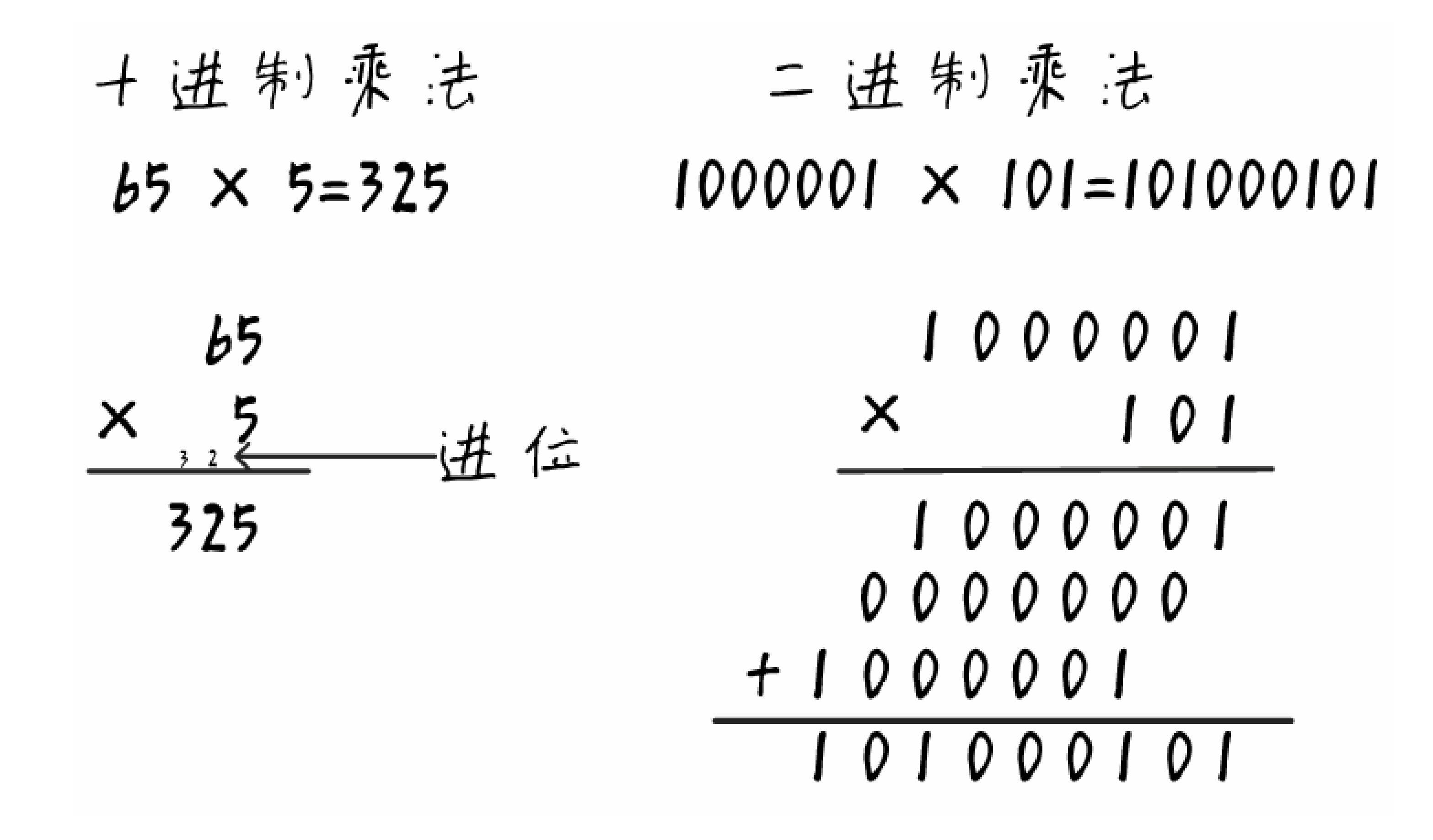


图 1-31

4．除法

除法是乘法的逆运算，既然二进制的乘法表只有 4 项，则其除法表也对应有如下 4 项（其中除以 0 是无意义的）：

0 ÷ 0 （无意义）
0 ÷ 1=0
1 ÷ 0 （无意义）
1 ÷ 1=1

看看图 1-31 中乘法的逆运算，如图 1-32 所示。

二进制除法
101000101 ÷ 101=1000001

```
          1000001
    101)101000101
       -101
          0000101
           -  101
                0
```

图 1-32

从图 1-32 中可看出，二进制的除法运算也可简单地转化为“减法与移位”操作。

可以看出，在二进制中，加、减、乘、除算法都可转换为加法运算。对于减法运算，

只需要将被减数设置为负数，就可将其转换为加法；对于乘法，则可使用“加法与移位”操作来完成；对于除法，则可使用“减法与移位”操作来完成。

既然比较复杂的乘法和除法运算能简单地转化为加、减法和移位操作，因此，电脑中就只需要设计一个加法器即可，这样就简化了电路设计。

1.3.4　二进制数的分解

在前面的二进制计数规则中曾说过，二进制数从右向左依次为 1 位、2 位、4 位、8 位……，这是指二进制数中各位的意义。虽然只有 0 和 1 这两个数码，但是其所处的位置不同，表示数据的大小也不同。例如，有以下二进制数：

$$1011$$

这个二进制数共有 4 位，由 3 个 1 和 1 个 0 组成。同样的 3 个 1，由于其处于不同的位置，这些 1 所表示的大小也是不同的，其所处的位置称为权。按从右向左的顺序各位的含义如下：

- 第 1 个 1 表示“1 的个数”；
- 第 2 个 1 表示“2 的个数”；
- 第 3 个 0 表示“4 的个数”；
- 第 4 个 1 表示“8 的个数”。

因此，二进制数 1011 由 1 个 8、0 个 4、1 个 2、1 个 1 组成。按各位的权列出的算式如下：

$$\begin{aligned}(1011)_2 &= 1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0\\ &= 1\times 8+0\times 4+1\times 2+1\times 1\\ &= 11\end{aligned}$$

从这种按权展开式可看出，每个位的“权”表现为 2 的幂次关系，即相邻两位相同数码代表的值为 2 倍的关系（从右向左看，第 2 位的 1 是第 1 位的 1 的 2 倍）。

这种按权展开式可方便地将二进制数转换为十进制数。

1.3.5　十进制数转换为二进制数

将二进制数按权展开，就可以方便地将其转换为十进制数。那么，十进制数该怎么转换成二进制数呢？

十进制整数转换为二进制整数通常采用“除 2 取余，逆序排列”法。

具体做法是：用 2 整除十进制整数，可以得到一个商和余数；再用 2 去除商，又会得到一个商和余数，如此进行，直到商为 0 时为止，然后把先得到的余数作为二进制数的低位有效位，后得到的余数作为二进制数的高位有效位，依次排列起来。

例如，将十进制数 14 转换为二进制数时，“除 2 取余”的转换过程如图 1-33 所示。

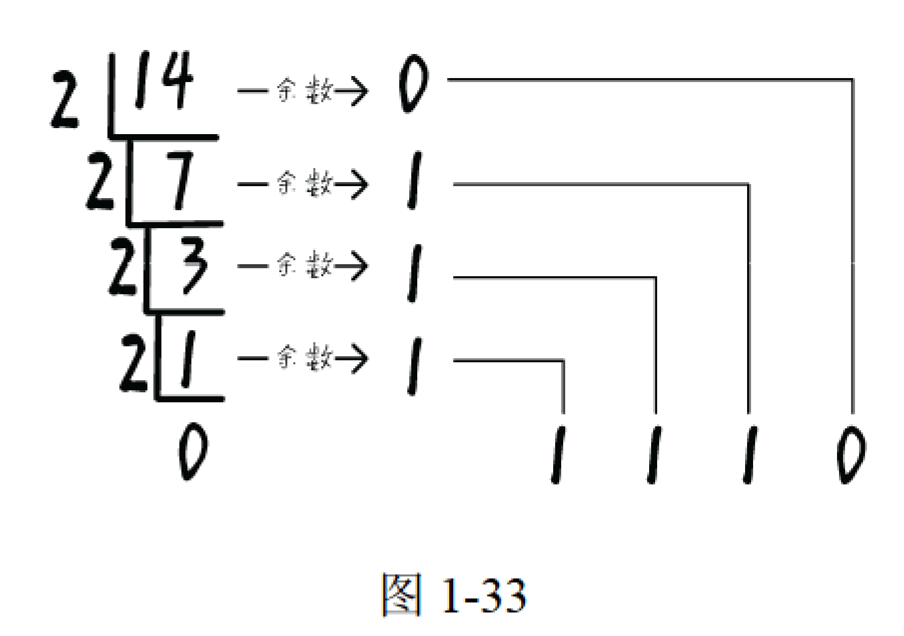


图 1-33

通过图 1-33 所示的逐项除 2 取余方式，得到每次除以 2 的余数，最后将取得的余数按逆序排列，即将十进制数 14 转换为二进制数 1110。将这个二进制数按权展开，又可得到十进制数 14。

$$(1110)_2=1\times 2^3+1\times 2^2+1\times 2^1+0\times 2^0$$

$$=1\times 8+1\times 4+1\times 2+0\times 1$$

$$=14$$

1.4 还有哪些进制

其实，除了我们常用的十进制数和电脑用的二进制数之外，生活中还有很多的计数进制，并且有很多的进制也在电脑中使用。

1.4.1 神奇的八卦：八进制

八卦最初是上古人们记事的符号，后被用为卜筮符号，古代常用八卦图作为除凶避灾的吉祥图案。因此，八卦也就被打上了封建迷信的标记。

1. 从八卦说起

其实，八卦中隐含了二进制和八进制的概念。首先，八卦的最基本概念是阴和阳，这就相当于二进制中的 0 和 1。在八卦图中用一根长实线代表阳，用一根中间断开的线代表阴，然后由 3 个这样的线条符号组成 8 种形式（相当于 3 位二进制数，可以表示 8

种状态），因此叫做八卦，如图 1-34 所示。

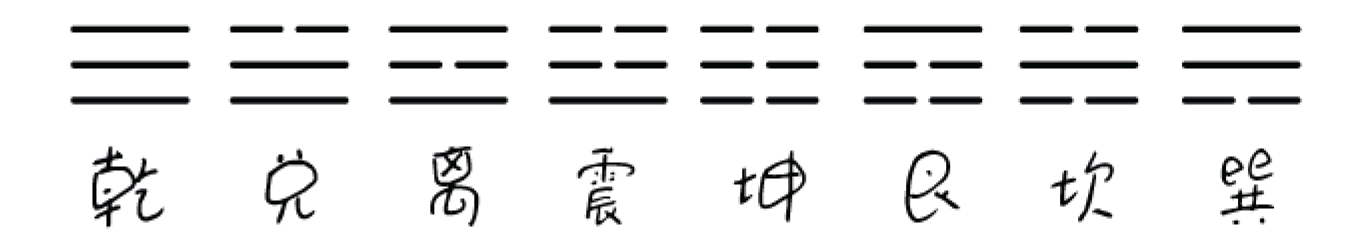


图 1-34

在八卦中，每一卦形代表一定的事物。乾代表天、坤代表地、坎代表水、离代表火、震代表雷、艮（gèn）代表山、巽（xùn）代表风、兑代表泽。

经过几千年的发展，八卦被赋予了很多的含义，除了上面介绍的代表自然现象之外，还可以代表方位、家族、五行，还可以将卦图转换为二进制数。如表 1-2 所示就是八卦中各卦所代表的不同含义。

表 1-2　八卦的含义

八卦名	八卦图像	自然	方位	二进制
乾	☰	天	西北	111
兑	☱	泽	西	110
离	☲	火	南	101
震	☳	雷	东	100
巽	☴	风	东南	011
坎	☵	水	北	010
艮	☶	山	东北	001
坤	☷	地	西南	000

2．一种计算方式：八进制

可以看出，八卦中的每一卦由 3 位二进制组成，这样表示 8 种状态的数据就是一种八进制计数方法。当然，八进制计数不会使用八卦的方式来表示，更多的情况下是使用阿拉伯数字来表示。

八进制计数法则主要有以下 3 个特点：

- ❑ 基数为 8；
- ❑ 由 8 个数码组成，分别是 0、1、2、3、4、5、6、7；
- ❑ 逢 8 进 1，借 1 当 8。

如表 1-3 所示是十进制数、八进制数和二进制数的对应关系。

表 1-3　十进制数、八进制数、二进制数对应关系

十进制	八进制	二进制
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100

续表

十进制	八进制	二进制
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010

从表 1-3 中可看出，1 位八进制数与 3 位二进制数相对应。

八进制计数法在早期的计算机系统中很常见，因此，偶尔我们还能看到人们使用八进制表示法。八进制适用于 12 位和 36 位计算机系统（或者其他位数为 3 的倍数的计算机系统）。但是，对于位数为 2 的幂（8 位、16 位、32 位与 64 位）的计算机系统来说，八进制就不算很好了。因此，在过去几十年里，八进制渐渐地淡出了电脑。不过，还是有一些程序设计语言提供了使用八进制符号来表示数字的能力，而且还是有一些比较古老的 UNIX 应用仍在使用八进制。

1.4.2 钟表使用的十二进制

另一个我们常用的进制就是十二进制，如图 1-35 所示的钟表表面显示为 12 个小时，即每 12 小时绕一圈，又从 0 点开始。



图 1-35

历史上，在很多古老文明中都使用十二进制来记数。这或许是由于一年中月球绕地球转 12 圈。在中国文化中，十二进制在记时中也有广泛应用。中国古代设有 12 地支，与一天的 12 个时辰对应。一个地支还对应 2 个节气，从而表示 1 年的 24 节气。同时，将地支与 12 种动物对应，成为 12 生肖，表示 12 年为周期的循环。

十二进制在各种度量衡中也经常会用到。如英制单位中 1 英尺等于 12 英寸，金衡制中 1 金衡磅等于 12 金衡盎司。

十二进制计数法的规则如下：

- 基数为 12；

- ❑ 由 12 个数码组成，分别是 0、1、2、3、4、5、6、7、8、9、A、B；
- ❑ 逢 12 进 1，借 1 当 12。

虽然十二进制在日常生活中很常用，不过，在计算机程序设计中使用的频率倒不多，反而是二进制、八进制、十进制、十六进制使用得要多一些。

1.4.3　半斤八两：十六进制

再来看一个计算机中使用得很多的进制：十六进制。

在日常生活中，也有很多使用十六进制的地方。中国原来使用的重量单位就是十六进制的，即 16 两为 1 斤，这也就有了所谓的“半斤八两”的说法了。

现在还在使用的磅和盎司这两个重量单位也是采用十六进制的方式计数的，1 磅等于 16 盎司。

计算机中使用二进制可以很好地计数和运算，为什么还要使用十六进制呢？

在使用二进制书写程序时，会发现有一个很麻烦的问题，要用二进制数表示一个比较大的十进制数时会需要很多位的二进制数。例如，表示十进制的 255，就需要 8 位二进制数，而表示 65535 这个十进制数，则需要 16 位二进制数。

$$(1111\ \ 1111)_2=(255)_{10}$$

$$(1111\ \ 1111\ \ 1111\ \ 1111)_2=(65535)_{10}$$

在程序中要处理的数据经常会比 65535 大得多，那就需要使用二进制数的更多位数了。对于很多位的二进制数，不但书写困难，还容易出错（这么长一串，稍不注意就可能输入错误）。

而使用十进制数又不方便与二进制数相对应，因此就引进了十六进制。

十六进制计数法的规则如下：

- ❑ 基数为 16；
- ❑ 由 16 个数码组成，分别是 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F；
- ❑ 逢 16 进 1，借 1 当 16。

如表 1-4 所示是十进制数、十六进制数和二进制数的对应关系。

表 1-4　十进制数、十六进制数、二进制数对应关系

十进制	十六进制	二进制
0	0	0
1	1	1
2	2	10
3	3	11

续表

十进制	十六进制	二进制
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

从表 1-4 中可看出，1 位十六进制数与 4 位二进制数相对应。这样，当程序员在编写程序时，若将数据按十六进制格式书写，将大大缩短输入数据的位数。例如，对于十进制数 255，用二进制表示需要 8 个 1 来表达（即 1111 1111），而用十六进制来表示，则只需要 2 个 F（即 FF）。类似地，对于十进制数 65535 用二进制来表示需要 16 位，而用十六进制来表示，则只需要 4 位。

$$(1111\ \ 1111)_2=(FF)_{16}$$

$$(1111\ \ 1111\ \ 1111\ \ 1111)_2=(FFFF)_{16}$$

1.4.4 60 年一个甲子：六十进制

在中国，经常可以听到“甲子”这个概念，这是农历历法中的一个概念，每 60 年为一个甲子，以天干与地支两者经一定的组合方式搭配成 60 对，为一个周期。

这就是六十进制的一种使用。

在现实生活中，使用六十进制的地方也很多。不过，与其他进位制不同，六十进制在一般运算和逻辑中并不常用，主要用于计算角度、地理坐标和时间。

例如，1 小时等于 60 分钟，而 1 分钟则为 60 秒。而 1 个圆可被均分成 360 度，每 1 度有 60 角分，1 角分等于 60 角秒。

六十进制在计算机程序中使用得很少。

1.4.5 各种进制之间的转换

前面介绍二进制时，曾介绍了二进制数与十进制数之间的转换方法：二进制数转换为十进制数时，将二进制数“按权展开”，即可得到十进制数；而十进制数转换为二进制

数时，使用“除 2 取余，逆序排列”即可。

类似地，其他进制的数转换为十进制数时也可采用相似的方法，只是在“按权展开”时，使用各进制的基数即可。因此，可得到其他进制转换为十进制的统一按权展开式：

$$D=X_{n-1}\times B^{n-1}+X_{n-2}\times B^{n-2}+\cdots+X_1B^1+X_0B^0$$

在上面的统一算式中，D 表示转换后得到的十进制数，X_{n-1} 为 B 进制中从右向左数第 n 位数。例如，二进制数 1101 的按权展开式为：

$$\begin{aligned}(1101)_2&=1\times 2^{(4-1)}+1\times 2^{(3-1)}+0\times 2^{(2-1)}+1\times 2^{(1-1)}\\&=8+4+0+1\\&=13\end{aligned}$$

而十六进制 BC0D 的按权展开式为：

$$\begin{aligned}(BC0D)_{16}&=11\times 16^{(4-1)}+12\times 16^{(3-1)}+0\times 16^{(2-1)}+13\times 16^{(1-1)}\\&=45056+3072+0+13\\&=48141\end{aligned}$$

如果要将十进制数转换为 B 进制数，也可以采用除以基数 B 再取余的方法来求得，如图 1-33 所示，只是将除数由 2 改为 B 进制数的基数 B 即可。

只要能将任意数转换为十进制数，然后又有将十进制数转换为任意进制数的方法，通过十进制数进行中转，即可进行任意进制数之间的转换了。使用这种思路可编写出以下的进制转换程序，可在任意两种进制之间进行转换。

```
#include<stdio.h>
#include<string.h>

//整数幂运算函数
int int pow(int x,int y)
{
    int i,result=1;
    for(i=1;i<=y;i++)
    {
        result*=x;
    }
    return result;
}

//将十进制数转换为任意进制，参数 dnum 为十进制数，参数 jz 为目标进制
void dtox(int dnum,int jz)
{
```

```
    char xnum[100];                     //保存目标进制的各位数
    int i=0,j=0;
    while(dnum>=jz)                     //用除模取余法计算对应进制的各位
    {
        if(dnum%jz<=9)
            xnum[j++]=dnum%jz+48;           //0～9 之间的数
        else
            xnum[j++]=dnum%jz-10+'A';       //超过 9 的数用大写字母表示
        dnum=dnum/jz;
    }
    if(dnum<=9)
        xnum[j]=dnum+'0';                   //0～9 之间的数
    else
        xnum[j]=dnum-10+'A';                //超过 9 的数用大写字母表示
    for(i = j;i>=0;i--)
    {
        printf("%c",xnum[i]);
    }
 }

/*将输入的数转换为十进制
参数 num 是一个数组，保存输入的字符串，参数 jz 为源数据的进制 */
int xtod(char num[],int jz)
{
      int dnum = 0,i,n=0,b;
      for(i=0;;i++)
      {
          if(num[i]=='\0')break;
          else n++;
      }
      for(i=n-1;i>=0;i--)                        //按权展开，转换为十进制
      {
          if(num[n-i-1]>='a')                    //小写字母
              b=num[n-i-1]-'a'+10;
          else if(num[n-i-1]>='A')               //大写字母
              b=num[n-i-1]-'A'+10;
          else                                   //数字
              b=num[n-i-1]-'0';
          dnum =dnum + b*int_pow(jz,i);
      }
      return dnum;                               //返回十进制数
}

//主函数
int main()
{
    char num[100];                               //保存要转换的数
    int jz1,jz2;                                 //保存两种进制
    printf("输入要转换的数：");
```

```
    scanf("%s",num);
    printf("输入数的进制: ");
    scanf("%d",&jz1);
    printf("要转换的进制: ");
    scanf("%d",&jz2);
    dtox(xtod(num,jz1),jz2);
    getch();
    return 0;
}
```

这个程序比较简单，定义了 4 个函数：

- main()函数为主函数，接收输入源数据、进制，以及需要转换成的进制。
- int_pow()函数为幂运算函数。在任意进制数按权展开时需要用到幂运算。
- xtod()函数为将任意进制转换为十进制的函数，通过按权展开的方式进行。
- dtox()函数为将十进制数转换为任意进制的函数，通过除模取余的方法进行。

编译运行以上程序，输入一个数据及其进制，然后输入要转换到的进制，即可得到结果。如图 1-36 所示，当输入 BC0D，并输入进制为 16，转换的进制为 10，则可得到 48141。



图 1-36

1.4.6　二进制与八进制、十六进制的转换

除了按以上方法进行任意进制之间的转换外，在程序中经常用到的是二进制与八进制、二进制与十六进制之间的转换。对于这些特殊进制间的转换，可以用更简单的方法。

二进制数转换为八进制数，可概括为“3 位并 1 位”的方法。具体转换方法是：按从右向左的方向，每 3 位二进制数为一组，最高位不足 3 位时，添 0 补足 3 位，然后将各组的 3 位二进制数按 2^2、2^1、2^0 权展开后相加，得到一位八进制数。例如：

$$(11010011101110011)_2=(151563)_8$$

001	101	001	101	110	011
1	5	1	5	6	3

而将八进制数转换为二进制数时，可采用相反的操作，即将每位八进制数拆为 3 位二进制数，称为“1 位拆 3 位”。

类似地，由于 1 位十六进制数与 4 位二进制数对应，二进制数转换为十六进制数时就可按“4 位并 1 位”的方法进行，而十六进制数转换为二进制数则可按“1 位拆 4 位”的方法进行。

$$(1101001101110011)_2=(D373)_{16}$$

1101	0011	0111	0011
D	3	7	3

第 2 章　神奇的素数

素数在自然数中占有非常重要的地位，素数是一类既简单又神秘的数字。说其简单，是因为小学生也知道什么是素数；说其神秘，是因为从古至今，多少数学家都想弄明白素数的规则，却一直没有找到其分布规律。

数学家都没研究出来的规律，程序员当然也不可能会找到。但是，任何事物都有正反两面，正是由于素数的无规律特点，在密码学中就可以大量采用。另外，在一些齿轮啮合设计中，也通常将齿轮的齿数设计成素数，以增加两齿轮中两个相同的齿相遇啮合次数的最小公倍数，这样可增强齿轮的耐用度，减少故障。

2.1　怎么判断素数

怎么判断素数呢？首先需要对素数进行定义，然后根据其定义判断指定的数是不是素数。对程序员来说，可以按素数的定义编写相应程序对素数进行判断。

2.1.1　什么是素数

数学中的定义是这样的：素数，又称为质数，是指在一个大于 1 的自然数中，除了 1 和此整数自身外，无法被其他自然数整除的数。或者说素数是只有 1 和本身两个因数的数。

比 1 大但不是素数的数称为合数。1 和 0 既非素数也非合数。

根据素数的定义，可用如图 2-1 所示方式列出 10 以内各数的因数，从而得出 2、3、5、7 为素数，而 4、6、8、9 不是素数。

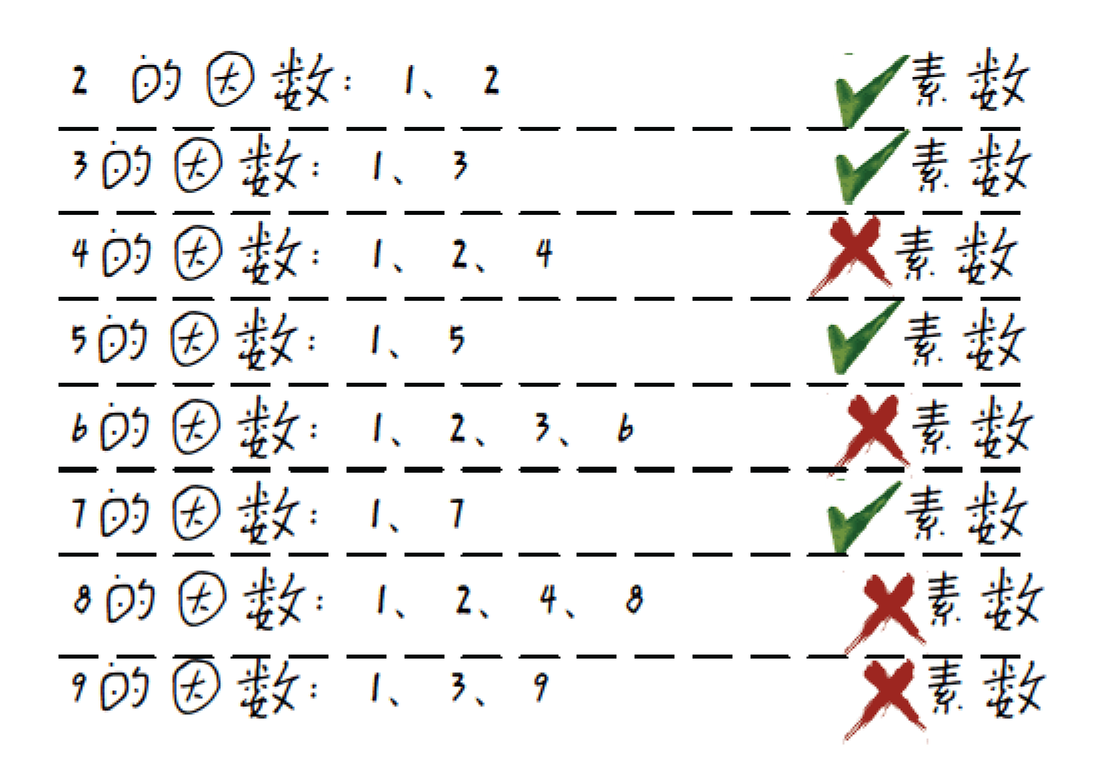


图 2-1

从上面的因数分解可看出，10 以内共有 4 个素数。随着数据的增大，素数的分布频率将变得更稀少，10000 以内的素数共有 1229 个，由于篇幅所限不逐一列出，下面列出 1000 以内的 168 个素数，如图 2-2 所示。

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199

211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293

307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397

401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499

503 509 521 523 541 547 557 563 569 571 577 587 593 599

601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691

701 709 719 727 733 739 743 751 757 761 769 773 787 797

809 811 821 823 827 829 839 853 857 859 863 877 881 883 887

907 911 919 929 937 941 947 953 967 971 977 983 991 997

图 2-2

对图 2-2 所列出的 1000 以内的素数进行总结，可看出在以 100 为间隔的区间中素数的个数是没有规律的，其中：

- 100 以内的素数有 25 个；
- 100～200 之间的素数有 21 个；
- 200～300 之间的素数有 16 个；
- 300～400 之间的素数有 16 个；
- 400～500 之间的素数有 17 个；
- 500～600 之间的素数有 14 个；
- 600～700 之间的素数有 16 个；
- 700～800 之间的素数有 14 个；
- 800～900 之间的素数有 15 个；
- 900～1000 之间的素数有 14 个。

目前最大的已知素数是 $2^{57885161}-1$（此数字位长度是 17425170），它是在 2013 年 1 月 25 日由 GIMPS 发现的。该组织还在 2008 年 8 月 23 日发现了目前所知第二大的已知素数 $2^{43112609}-1$（此数字位长度是 12978189）。

2.1.2 验证素数

在图 2-2 中列出了 1000 以内的素数，究竟这些数据是不是素数呢？需要进行验证。

验证一个自然数是否为素数，这个问题在中世纪就引起了人们的注意，当时人们试

图寻找一个公式一劳永逸地解决问题，到了高斯时代，基本上确认了简单的素数公式是不存在的，因此，高斯认为对素性判定是一个相当困难的问题。从此以后，这个问题吸引了大批数学家。 验证素数的算法可分为两大类，即确定性算法及随机算法，如图 2-3 所示。

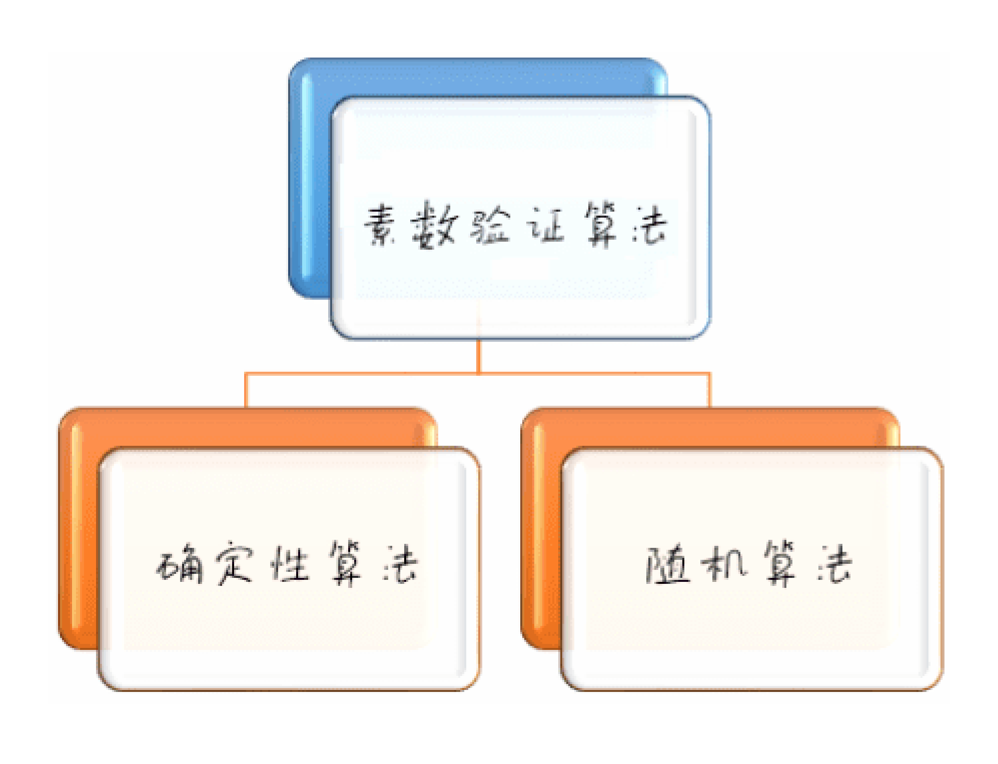


图 2-3

确定性算法可得出确定的结果，但通常算法较慢，而随机算法正好相反。通过计算机进行运算，可解决算法较慢的问题，其算法的实现也很简单。

确定性算法中最常用的就是试除法。试除法是根据素数的定义得出的一种方法，用需要验证的数 N 逐个除以从 2 开始至 N–1 中的所有数，若能被某一个数整除，表示有一个因数，说明数 N 不是素数；若直到 N–1 都不能被整除，则说明该数是素数。

根据以上思路，可编写以下的试除法函数：

```
int is prime(int n)
{
    int i;

    if(n <= 1)                          //1 不是素数，参数 n<=1 时返回 0
    {
        return 0;
    }

    for(i = 2; i < n; i++)
    {
        if(n % i == 0)                  //判断 n 是否能被 i 整除
        {
            return 0;
        }
    }

    return 1;
}
```

其实，可以对素数的定义进行进一步的分析。要判断数 N 是否为素数，不需要用 N 一直除到 N–1 才能确认，而只需要除到 $\sqrt{n}$ 就可以了。例如，N=15，则能整除 15 的除数有 1、3、5，对于除数 5 就不用判断，因为 N 被 3 整除时其商就是 5，也就表示 N 能被 5 整除了。

因此，为了减少循环判断的次数，提高程序的执行效率，可将函数 is_prime()进行修改，以提高程序的效率。

```
int is_prime(int n)
{
    int i;

    if(n <= 1)                              //1 不是素数，参数 n<=1 时返回 0
    {
        return 0;
    }

    for(i = 2; i * i <= n; i++)
    {
        if(n % i == 0)                      //判断是否能被 i 整除
        {
            return 0;
        }
    }

    return 1;
}
```

从上面的程序可看到，在 for 循环中以 i^2 与 n 值进行比较，就可以显著地减少循环的次数，从而提高验证的效率。

2.1.3 寻找素数的算法

通过验证方法可以验证某个整数是否为素数，而寻找素数的方法，就是寻找在给定限度内的所有素数排列。例如，要求出 1000 以内的所有素数，就是一个寻找素数排列的问题。由于已经有上面定义的 is_prime()函数，求出 1000 以内所有素数的方法就很简单了，可以用以下程序来完成。

```
#include <stdio.h>
int is prime(int n)
{
    int i;

    if(n <= 1)                              //1 不是素数，参数 n<=1 时返回 0
    {
        return 0;
    }

    for(i = 2; i * i <= n; i++)
    {
        if(n % i == 0)                      //判断是否能被 i 整除
        {
            return 0;
        }
    }

    return 1;
```

```
}

int main()
{
    int i,n=0,t=1;
    printf("1000 以内的素数排列：\n");

    for(i=2;i<1000;i++)
    {
        if(is_prime(i))
        {
            n++;
            t++;
            printf("%4d",i);                        //输出素数
            if(t>10)                                //每输出 10 个素数就换行
            {
                printf("\n");
                t=1;
            }
        }
    }
    printf("\n1000 以内的素数共有%d 个\n",n);

    getch();
    return 0;
}
```

执行以上程序，可得到 1000 以内的所有素数列表，如图 2-4 所示。

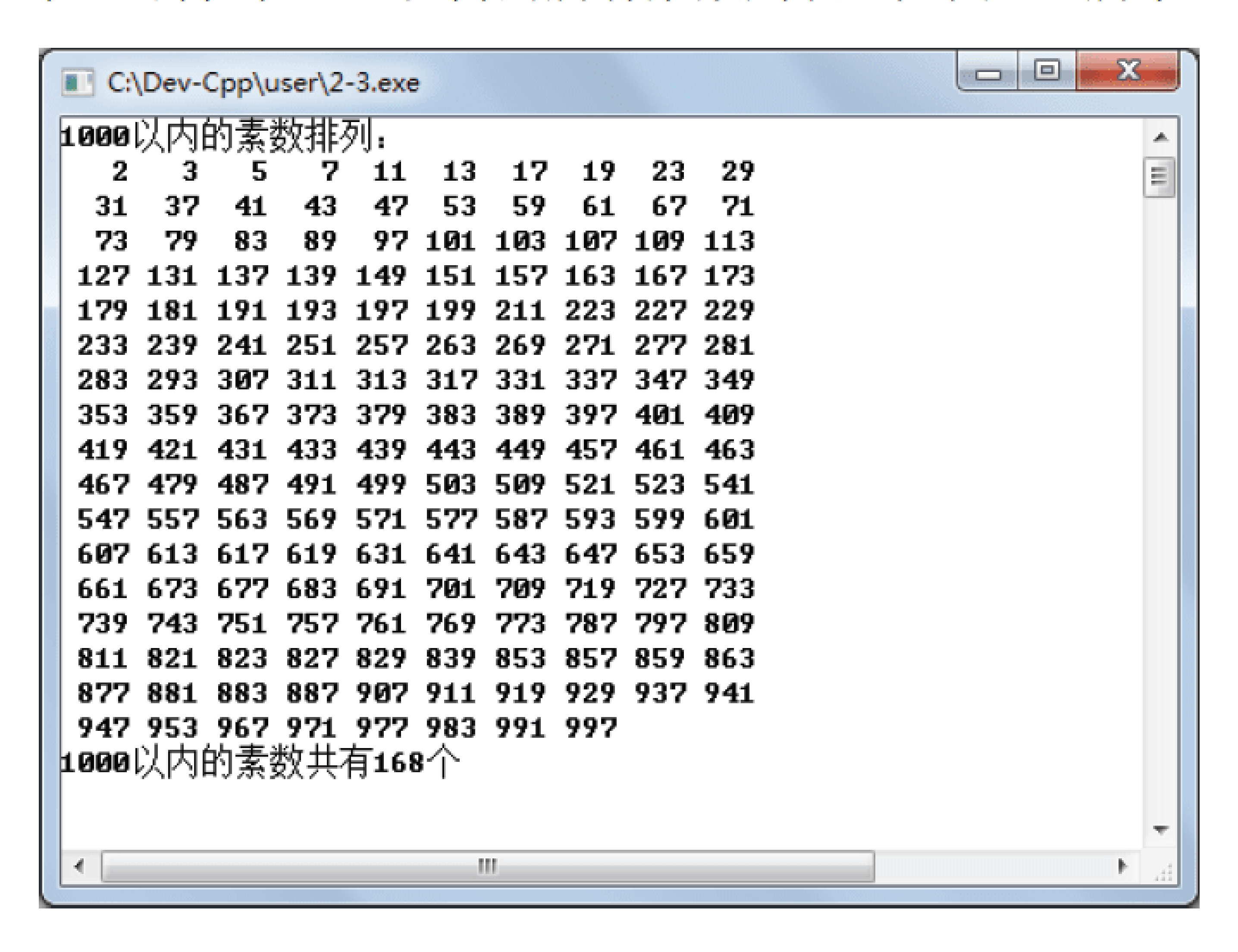


图 2-4

在上面的代码中，通过 is_prime()函数来验证指定区间（2～1000）中的每一个数是否为素数，而 is_prime()函数中又通过循环进行验证。这种双循环会导致程序执行效率不高。

这时可考虑采用另一种寻找素数的算法：著名的 Eratosthenes 求素数方法。下面演示如何用这种方法求 100 以内的素数。

Eratosthenes 算法假设有一个筛子，用来存放 2～100 之间的所有数，如图 2-5（a）所示。

由于偶数都能被 2 整数，因此偶数都不是素数（2 除外），则将这些数筛去，得到如图 2-5（b）所示的数据（设置了背景色的数字将被筛去）。

接着再将 3 的倍数筛去，得到如图 2-5（c）所示结果。

接下来继续将 5 的倍数筛去，得到图 2-5（d）所示结果。

最后再将 7 的倍数筛去，得到如图 2-5（e）所示结果，即可得到 100 以内的所有素数。

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

（a）

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

（b）

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

（c）

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

（d）

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

（e）

图 2-5

从图 2-5 中可看到，在使用 Eratosthenes 算法进行筛选时，只需要执行 4 次筛选就完成了 100 以内的素数的筛选，效率非常高。如果要筛选的数据范围更大，由于只需要选择已经筛选过的素数对后面的数进行筛选，因此可快速筛选出后面的素数。

从图 2-5 中的算法过程可以看出，Eratosthenes 算法比试除法的效率要高得多。

根据图 2-5 中所示的过程编写相应的程序，具体代码如下：

```
#include <stdio.h>
#define MAXNUM 1000                                //求 1000 以内的所有素数
int main()
{
    int i,j,c=0;
    int prime[MAXNUM+1];                           //定义保存素数的数组
    for(i=2;i<=MAXNUM;i++)                         //初始化数组
        prime[i]=1;                                //标志为 1 表示对应的数是素数

    for(i=2;i*i<=MAXNUM;i++)                       //循环处理前 i 个
    {
        if(prime[i]==1)                            //若为素数，则进行筛选
        {
            for(j=2*i;j<=MAXNUM;j++)               //筛去合数
            {
                if(!prime[j]) continue;            //是合数，则跳过
                if(j%i==0)                         //数 j 能被整除，说明不是素数
                    prime[j]=0;                    //清除标志
            }
        }
    }

    for(i=2;i<MAXNUM;i++)                          //输出素数
    {
        if(prime[i]==1)                            //是素数，则输出
        {
            printf("%4d ",i);                      //输出素数
            c++;
            if(c%10==0)                            //每行输出 10 个素数
                printf("\n");
        }
    }

    printf("\n 共有%d 个素数！",c);
    getch();
    return 0;
}
```

2.1.4　已被证明的素数定理

自古以来就有很多数学家研究素数，因此，得出了许多与素数有关的定理。下面简

单介绍一些已被证明的素数定理。

1．在$(a,2a]$之间必有一个素数

在一个大于 1 的数 a 和它的 2 倍之间（即区间$(a, 2a]$中）必存在一个素数。如图 2-6 所示，可看到在$(a, 2a]$区间中都至少包含一个素数。

2．存在任意长度的素数等差数列

什么是等差数列呢？这是一个古老的数学课题。一个数列从第二项起，从后项减去前项所得的差是一个相同的常数，则这个数列就被称为等差数列。

2~4 中有素数3
3~6 中有素数5
4~8 中有素数5、7
5~10 中有素数7
6~12 中有素数7、11
7~14 中有素数11、13
50~100 中有素数53、59、61……

图 2-6

等差数列：1、3、5、7

用素数构成的等差数列被称为素数等差数列。例如从 5 开始，以 12 为间隔常数，就可以得到这样的序列：

5、17、29、41、53、65•••

对这个数列来说，只有前 5 个数是素数，第 6 个数 65 能被 5 整除，不是素数，因此，在这里得到的是由 5 个素数构成的素数等差数列：

5、17、29、41、53

还有更长的素数等差数列吗？当然有，如下面的 10 个素数就构成间隔为 210 的素数等差数列：

199、409、619、829、1039、1249、1459、1669、1879、2089

2004 年 4 月 18 日，格林和陶哲轩两人宣布：他们证明了“存在任意长度的素数等差数列”，也就是说，对于任意值 *K*，存在 *K* 个成等差级数的素数。例如 *K*=3，有素数序列 3、5、7（两数之间差 2）……*K*=10，有素数序列 199、409、619、829、1039、1249、1459、1669、1879、2089（两数之间差 210）。他们将长达 50 页的论文——《素数含有任意长度的等差数列》张贴在当日的预印本网站上，并向《美国数学年鉴》（*Annals of Mathematics*）投稿。

这是一项惊人的成就，他们的发现揭示了素数中存在的某种规律。他们的证明立即在国际学术界引起轰动。

3．其他已证明的素数定理

已证明的素数定理还包括以下几项：

- 一个偶数可以写成两个数字之和，其中每一个数字最多只有 9 个质因数。
- 一个偶数必定可以写成一个质数加上一个合数，其中的因子个数有上界。
- 一个偶数必定可以写成一个质数加上一个最多由 5 个因子所组成的合数。后来，有人简称这个结果为（1+5）。
- 一个充分大偶数必定可以写成一个素数加上一个最多由 2 个质因子所组成的合数，简称为（1+2）。

2.2　孪生素数

我们知道，素数在自然数中的比例很少，而孪生素数就更少了。那么，什么是孪生素数？孪生素数有什么特点呢？

2.2.1　什么是孪生素数

所谓孪生素数，是指间隔为 2 的相邻素数，它们之间的距离已经近得不能再近了，就像孪生兄弟一样，因此被称为孪生素数，也称为双生素数。

例如，素数 3 和 5，其间距为 2，就是一组孪生素数。100 以内的孪生素数还有 5 与 7，11 与 13，17 与 19，29 与 31，41 与 43，59 与 61，71 与 73。

(3 , 5)　(5 , 7)　(11 , 13)　(17 , 19)

(29 , 31)　(41 , 43)　(59 , 61)　(71 , 73)

100 以内的孪生素数共有 8 对，不过，随着数字的增大，孪生素数的分布变得越来越稀疏，寻找孪生素数也变得越来越困难。那么会不会在超过某个界限之后就再也不存在孪生素数了呢？

2.2.2 孪生素数的公式

对于自然数 Q 与 Q+2，若都不能被小于 $\sqrt{Q+2}$ 的任何素数整除，则 Q 与 Q+2 就构成一对孪生素数。这句话可以用以下公式表达：

$$
\begin{aligned}
Q &= P_1M_1+B_1 \\
&= P_2M_2+B_2 \\
&= \cdots\cdots \\
&= P_KM_K+B_K
\end{aligned}
$$

以上公式中，P_1、P_2、P_k 表示从小到大的顺序素数 2、3、5、7……，$B_i \neq 0$ 且 $B_i \neq P_i-2$，若 $Q<(P_{K+1})^2-2$，则 Q 与 Q+2 是一对孪生素数。也就是说，将数 Q 分解后，最小剩余不能为 0 和 P_i-2，例如 Q 不能是 $2M$，$3M+1$，$5M+3$，$7M+5$……，P_iM_i-2，否则 Q+2 就是合数。

看一个例子吧。假设 Q=29，可分解为以下算式：

$$
\begin{aligned}
29 &= 2\times 14+1 \quad (2M+1) \\
&= 3\times 9+2 \quad (3M+2) \\
&= 5\times 5+4 \quad (5M+4)
\end{aligned}
$$

由于 $\sqrt{29+2}$ 的结果取整后为 5，因此，在上式中只按公式分解到 5 为止，一共有 3 项（即 k=3），每项的 B 值都不为 0，且 B_i 的每一项不等于 P_i-2。因此，可得到 29 与 29+2 是一对孪生素数。

上式也可使用同余式来表达：

$$
\begin{aligned}
Q \div 2 &= 14 \text{ 余 } 1 \\
Q \div 3 &= 9 \text{ 余 } 2 \\
Q \div 5 &= 5 \text{ 余 } 4
\end{aligned}
$$

根据中国剩余定理，对于给定的余数，在除数为素数的范围内有唯一的解。例如在上式中，除数为 2、3、5 时余数也知道了，因此就可以推算出 Q 的值为 29。

2.2.3 中国剩余定理

什么是中国剩余定理呢？先来看一则中国民间的传说故事——“韩信点兵”。

秦朝末年，楚汉相争。一次，韩信将 1500 名将士与楚王大将李锋交战。苦战一场，楚军不敌，败退回营，汉军也死伤四五百人，于是韩信整顿兵马也返回大本营。当行至

一山坡，忽有后军来报，说有楚军骑兵追来。只见远方尘土飞扬，杀声震天。汉军本来已十分疲惫，这时队伍大哗。韩信兵马到坡顶，见来敌不足五百骑，便急速点兵迎敌。他命令士兵 3 人一排，结果多出 2 名；接着命令士兵 5 人一排，结果多出 3 名；他又命令士兵 7 人一排，结果又多出 2 名。韩信马上向将士们宣布：我军有 1073 名勇士，敌人不足 500，我们居高临下，以众击寡，一定能打败敌人。汉军本来就信服自己的统帅，这一来更相信韩信是“神仙下凡”、“神机妙算”。于是士气大振。一时间旌旗摇动，鼓声喧天，汉军步步紧逼，楚军乱作一团。交战不久，楚军大败而逃。

韩信是怎么知道军队有 1073 名士兵的呢？

对于这个问题，可将其描述为一个数学问题，就是：一个数除以 3 余 2，除以 5 余 3，除以 7 余 2，求这个数是多少？

先列出除以 3 余 2 的数：

2，5，8，11，14，17，20，23，26，…

再列出除以 5 余 3 的数：

3，8，13，18，23，28，…

在这两列数中，首先出现的公共数是 8，3 与 5 的最小公倍数是 15，两个条件合并成一个就是：

$$8+15\times n$$

将 n 分别取 1、2、3、…即可得到数列：

8，23，38，…

再列出除以 7 余 2 的数：

2，9，16，23，30，…

可以看出，符合题目条件的最小数是 23。

也就是说，我们已把题目中三个条件合并成一个：该数除以 105 余 23。

由于汉军原有士兵 1500 人，死伤四五百人，即剩余的士兵应为 1000 余人，即可得到士兵的总数：

$$105\times 10+23=1073$$

2.2.4　孪生素数分布情况

我们知道，素数本身的分布是随着数字的增大而越来越稀疏，不过幸运的是，早在古希腊时代，欧几里得（Euclid）就证明了素数有无穷多个。长期以来人们猜测孪生素

数也应该有无穷多组，可是该怎么验证这个猜想呢？

对于程序员来说，当然是想编写程序来查找素数和孪生素数。当然，由于计算机位数的限制，所表示的整数范围是有限的，如果要找出更多的素数或孪生素数，需要另外编写大整数处理的相关功能。

由于篇幅限制，本书将不介绍大整数方面的功能，下面只列出一个简单的求解孪生素数的程序。在程序中，将先筛选出素数，然后在素数中再筛选出孪生素数。

```
#include <stdio.h>
#define MAXNUM 10000                          //求 10000 以内的所有素数
int main()
{
    long i,j,c=0,twin=2,t=0;
    char prime[MAXNUM+1];                     //定义保存素数的数组
    prime[0]=0;
    prime[1]=0;
    for(i=2;i<=MAXNUM;i++)                    //初始化数组
        prime[i]=1;                           //标志为 1 表示对应的数是素数

    for(i=2;i*i<=MAXNUM;i++)                  //筛选合数
    {
        if(prime[i]==1)                       //若为素数
        {
            for(j=2*i;j<=MAXNUM;j++)          //筛去合数
            {
                if(!prime[j]) continue;
                if(j%i==0)                    //数 j 能被整除，说明不是素数
                    prime[j]=0;               //清除标志
            }
        }
    }

    //统计素数数量
    for(i=2;i<MAXNUM;i++)
    {
        if(prime[i]==1)
        {
            c++;
            if(i-2==twin)                     //是孪生素数
            {
                printf("(%d,%d) ",twin,i);
                t++;
                if(t%5==0)  printf("\n");
            }
            twin=i;
        }
    }

    printf("\n 共有%d 个素数，%d 对孪生素数！",c,t);
    getch();
    return 0;
}
```

执行以上程序，将得到如图 2-7 所示的结果。从结果中可以看到，在 10000 以内共

有 1229 个素数，而孪生素数只有 205 对。

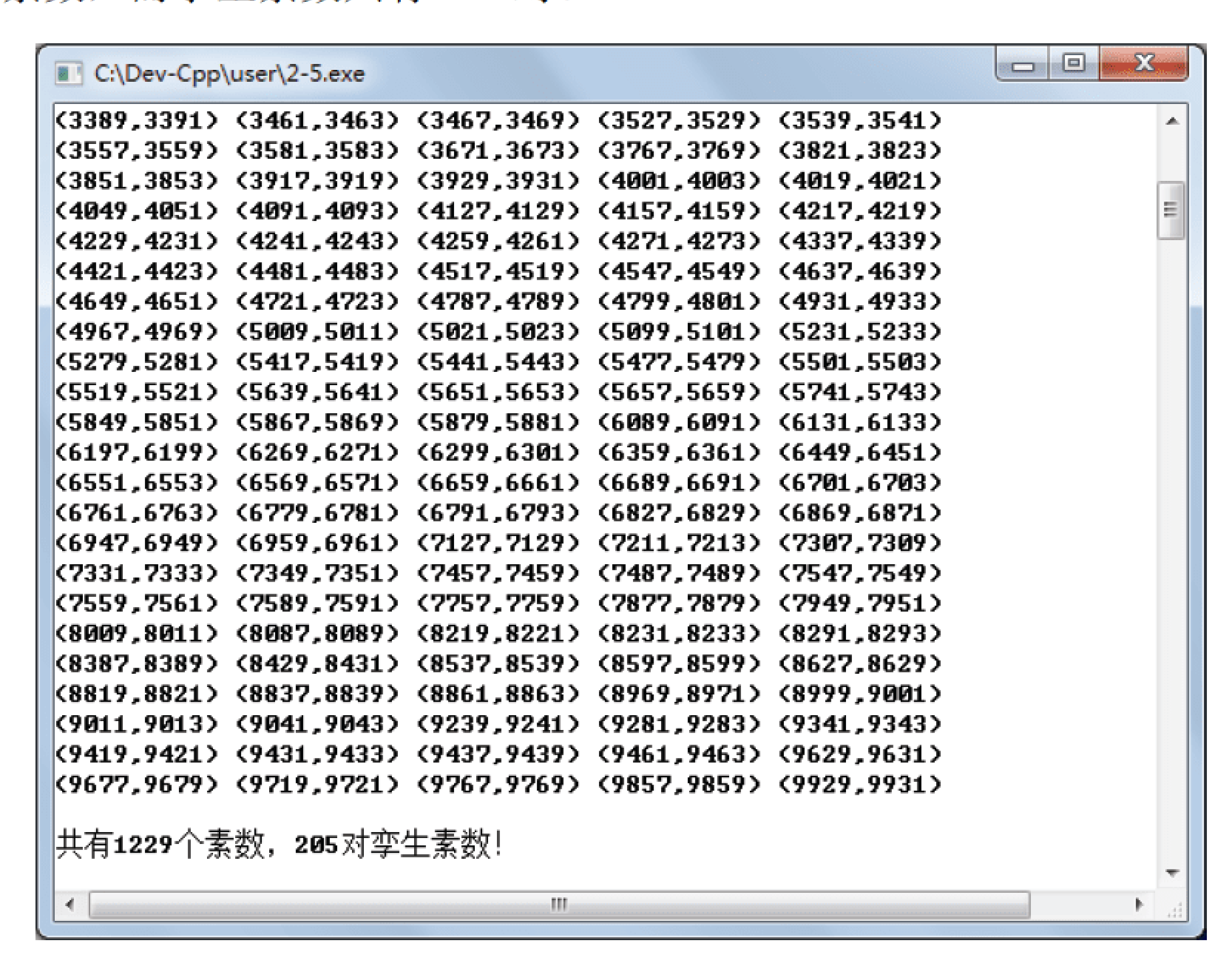


图 2-7

在上面的程序中，常量 MAXNUM 定义为 10000，即可求出 10000 以内的素数，如果将 MAXNUM 定义为更大的常量值，则可求出更多的素数和孪生素数。当然要注意，由于计算机中变量表示范围及程序栈空间大小的限制，MAXNUM 定义的数据大小是有限的。

2.3　使用素数的 RSA 算法

RSA 公钥加密算法是第一个既能用于数据加密也能用于数字签名的算法。它易于理解和操作，也十分流行。RSA 算法就是素数的典型应用。下面简单介绍一下 RSA 算法，要完整地实现 RSA 算法，需要较长的代码，本书就不给出相应的程序了，而是重点介绍 RSA 的概念、原理和实现的过程。

2.3.1　什么是 RSA

在计算机中常用的加解密技术分为两类，即对称加密和非对称加密。

在对称加密技术中，对信息的加密和解密都使用相同的密钥 Key，如图 2-8 所示，也就是说使用同一个密钥 Key 对数据进行加密和解密。这种加密方法可简化加解密的处理过程，信息交换双方都不必彼此研究和交换专用的加解密算法。如果在交换阶段，密钥 Key 没有泄露，那么加密数据的机密性和报文完整性就可以得到保证。

对称加密技术虽然简单，但存在一些不足，由于加密、解密都需要使用同一个 Key，这样，信息传送双方都要接触到这个 Key，密钥 Key 更容易泄露。

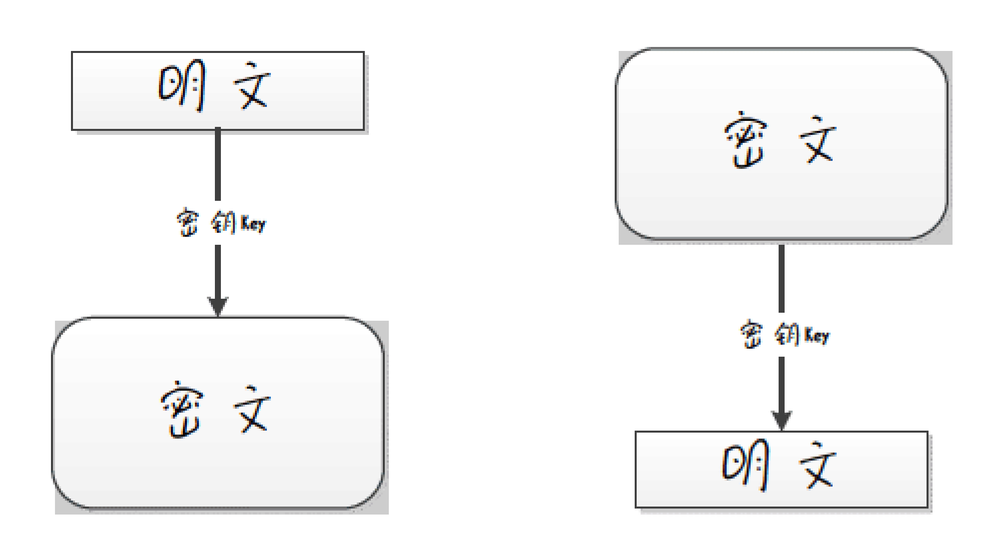


图 2-8

而非对称加密（又称为公开密钥加密）中，不再只有一个密钥 Key 了。在非对称加密算法中，密钥被分解为一对，一个称为公开密钥（简称公钥 PK），另一个称为私有密钥（简称私钥 SK）。对于公钥，可以通过非保密方式向他人公开，而私钥则由解密方保存，不用对别人公开。

发送信息的一方通过公钥对数据进行加密，然后发送给接收方。接收方通过私钥对密文进行解密，加、解密的过程如图 2-9 所示。

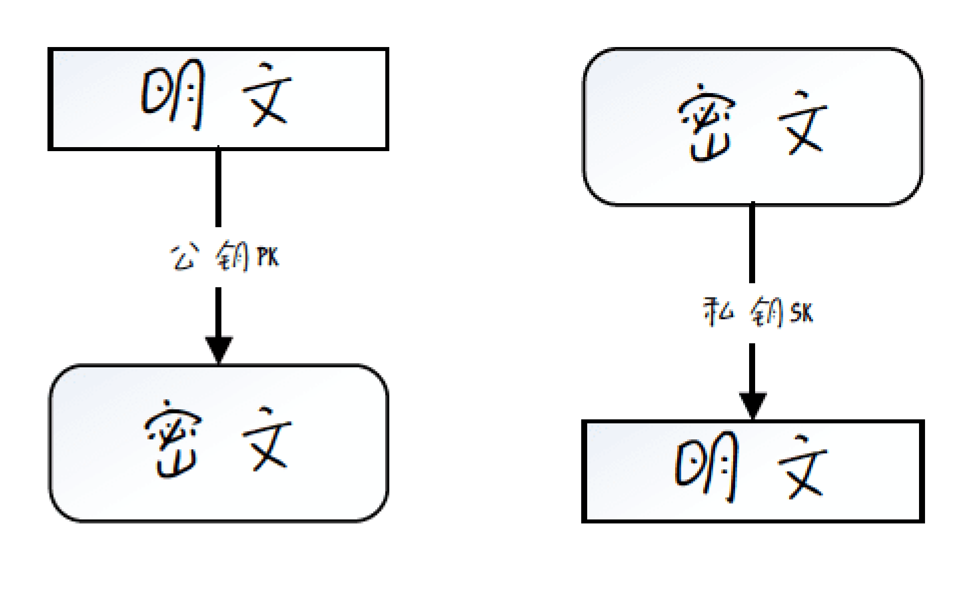


图 2-9

由于非对称加密方式可以使通信双方无须事先交换密钥就可以建立安全通信，因此被广泛应用于身份认证、数字签名等信息交换领域。

非对称加密体系一般是建立在某些已知的数学难题之上，是计算机复杂性理论发展的必然结果。最具有代表性的非对称加密方式是 RSA 公钥密码体制。

2.3.2　RSA 算法基础

在 RSA 算法中，最基础的一个定理就是 RSA 定理，这个定理描述如下：

若 P 和 Q 是两个相异质数，另有正整数 R 和 M，其中 M 的值与 $(P-1)(Q-1)$ 的值互质，并使得 $(RM) \bmod (P-1)(Q-1) \equiv 1$。有正整数 A，且 $A < PQ$，设：

$$C \equiv A^R \mod PQ$$

$$B \equiv C^M \mod PQ$$

则有：$A = B$

在以上描述中 mod 表示取余的运算。

在 RSA 算法中还引用了很多定理，详细介绍的话需要很大篇幅，这里就不逐一介绍了。下面介绍一下 RSA 算法的基础操作步骤。

1．生成公钥和私钥

生成公钥 PK 和私钥 SK 的步骤如下。

（1）随意选择两个大的素数 *P* 和 *Q*，*P* 不等于 *Q*。

（2）将 *P*、*Q* 两素数相乘得到一个数 *N*，即 $N=PQ$。

（3）将 *P*、*Q* 分别减 1，再相乘，得到一个数 *T*，即 $T=(P-1)(Q-1)$。

（4）选择一个整数 *E*，作为一个密钥，使 *E* 与 *T* 互质（即 *E* 与 *T* 的最大公约数为 1），并且 *E* 必须小于 *T*。

（5）根据公式 $DE \bmod T\equiv1$，计算出 *D* 的值，作为另一个密钥。

（6）通过以上步骤计算得出 *N*、*E*、*D* 这 3 个数据，其中（*N*、*E*）作为公钥，（*N*、*D*）作为私钥（当然也可以将公钥和私钥互换）。

（7）生成公钥和私钥后，就可以将公钥对外发布了，如图 2-10 所示。



图 2-10

2．用公钥加密信息

发送信息的一方收到公钥 PK 后，就可以通过公钥 PK 对数据进行加密。加密的操作步骤如下，其中明文为 M，加密后得到的密文为 C，公钥为（N，E）。

明文：M

加密：$M^E \bmod N \equiv C$

密文：C

3．用私钥解密信息

接收方持有私钥（N，D），在接收到密文 C 后，即可通过私钥进行解密，得到明文 M。解密的过程如下：

密文：C

解密：$C^D \bmod N \equiv M$

明文：M

2.3.3 RSA 算法实践

了解 RSA 算法生成密钥、加密、解密的过程之后，接下来进行一次 RSA 算法的模拟操作，进一步了解 RSA 算法的使用过程。

1．生成公钥和私钥

生成公钥 PK 和私钥 SK 的过程如下：

取 $P=11,\ Q=13$

则 $N=PQ=11\times 13=143$

$T=(P-1)(Q-1)=10\times 12=120$

取 $E=7$

由 $D\times E \bmod T \equiv 1$

$D\times 7 \bmod 120 \equiv 1$

得 $D=103$

则：公钥(143,7)，私钥(143,103)

2．用公钥加密

有了公钥，就可方便地进行数据加密了。在上面这个例子中的公钥为（143，7），私钥为（143，103），由于是手工计算，为了使计算的数据量小一点，因此将公钥与私钥进行交换，即公钥使用（143，103），而私钥则使用（143，7）。设明文为 2，则加密过程如下：

$$
\begin{aligned}
\text{明文：} M &= 2 \\
\text{加密：} C &= M^E \bmod N \\
&= 2^{103} \bmod 143 \\
&= 10141204801825835211973625643008 \bmod 143 \\
&= 63 \\
\text{密文：} C &= 63
\end{aligned}
$$

3．用私钥解密

收到密文 C（这里为 63）后，则可以通过私钥（143，7）进行解密，解密过程如下：

$$
\begin{aligned}
\text{密文：} C &= 63 \\
\text{解密：} M &= C^D \bmod N \\
&= 63^7 \bmod 143 \\
&= 3938980639167 \bmod 143 \\
&= 2 \\
\text{明文：} M &= 2
\end{aligned}
$$

从以上生成密钥、加密、解密的过程可以看出，虽然我们这里只使用了两个小的素数 11 和 13，但其计算量却很大，特别是在加密和解密过程中，需要进行幂运算，得到的结果将是一个非常大的整数。在实际应用中，P、Q 要取很大值的素数，则得到的 N、D、E 值也将很大，所以在加密、解密过程中的幂运算结果将更大，通过 C 语言（或其他程序设计语言）提供的基本数据类型已经没办法保存这么大的数了。

因此，在 RSA 算法中，虽然过程很简单，但需要编写程序处理大整数，包括大整数的加、减、乘、除、幂运算等。

2.3.4　RSA 应用：数字签名

RSA 的典型应用就是数字签名技术。

数字签名技术是实现交易安全的核心技术之一，它的实现基础就是 RSA 加密技术。在这里，我们介绍数字签名的基本原理。

以往的书信或文件是根据亲笔签名或印章来证明其真实性的。但在计算机网络中传送的报文又如何盖章呢？这就是数字签名所要解决的问题。数字签名必须保证以下几点：

- ❑ 接收者能够核实发送者对报文的签名；
- ❑ 发送者事后不能抵赖对报文的签名；

- 接收者不能伪造对报文的签名。

现在已有多种实现数字签名的方法，但采用公开密钥算法要比常规算法更容易实现。下面就来介绍这种数字签名。

发送者 A 用其私密密钥 SKA 对报文 M 进行运算，将结果 DSKA（M）传送给接收者 B。接收者 B 用已知的 A 的公开密钥得出 EPKA（DSKA（M））=M，如图 2-11 所示。

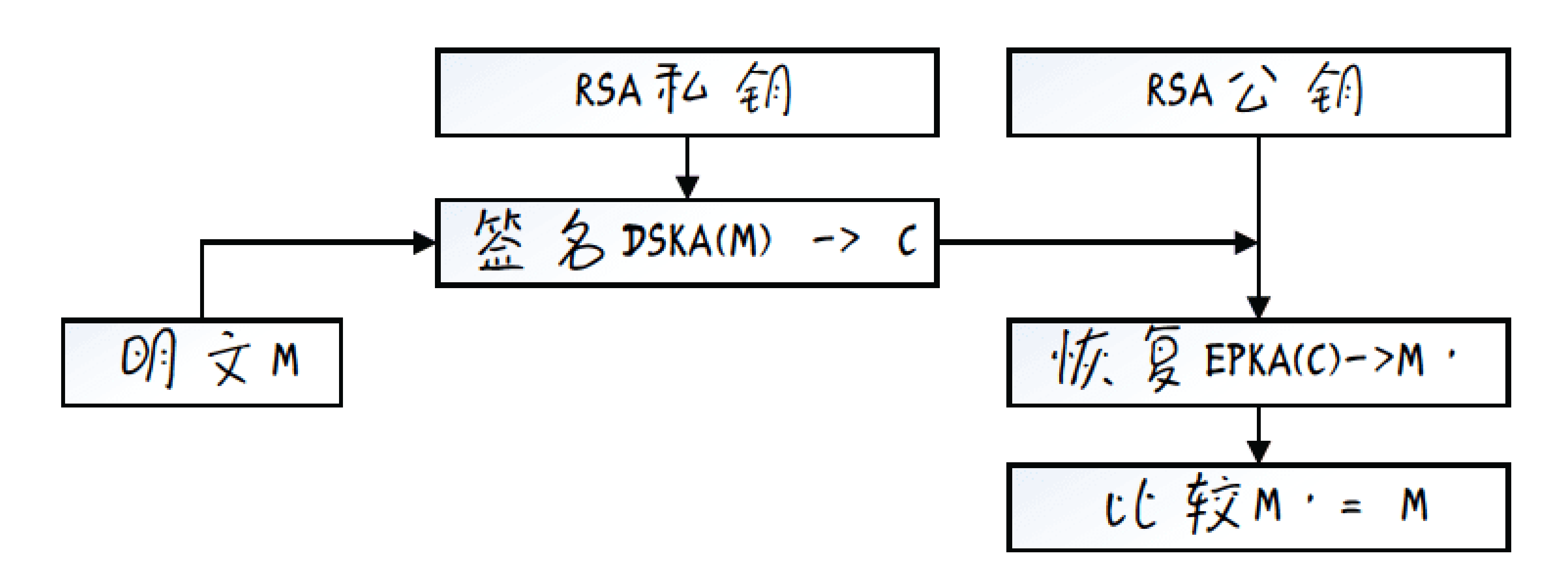


图 2-11

因为除 A 外没有别人能具有 A 的私密密钥 SKA，所以除 A 外没有别人能产生密文 DSKA（M）。这样，报文 M 就被签名了。用私钥加密后的密文发送给对方，对方只能用持有的公钥进行解密，以实现核实发送者对报文的签名。

假若用户 A 要抵赖曾经发送过报文给用户 B。用户 B 可将 M 及 DSKA（M）出示给第三方。第三方很容易用 PKA 去证实用户 A 确实发送消息 M 给用户 B。反之，如果是用户 B 将 M 伪造成 M'，则用户 B 不能在第三方面前出示 DSKA（M'）。这样就证明用户 B 伪造了报文。可以看出，实现数字签名也同时实现了对报文来源的鉴别。

2.3.5 RSA 被破解的可能性

加密与破解是一对矛盾，再强的加密方法，总有被破解的一天。RSA 算法是否安全呢？是否能被很轻松地破解呢？

RSA 是被研究得最广泛的公钥算法，从提出到现在经历了各种攻击的考验，逐渐为人们接受，被普遍认为是目前最优秀的公钥方案之一。

RSA 的缺点主要有以下两点：

- 产生密钥很麻烦，受到素数产生技术的限制，因而难以做到一次一密。
- 分组长度太大，为保证安全性，N 至少也要 600 比特二进制位以上，使运算代价很高，尤其是速度较慢，较对称密码算法慢几个数量级；且随着大数分解技术的发展，这个长度还在增加，不利于数据格式的标准化。目前，SET 协议中要求 CA 采用 2048 比特二进制位长的密钥，其他实体使用 1024 比特的密钥。

根据前面的运算过程可看出，RSA 算法的安全性依赖于大数分解。公钥和私钥都是两个大素数（大于 100 个十进制位）的函数。据猜测，从一个密钥和密文推断出明文的难度等同于分解两个大素数的积。

RSA 的安全性在于对于一个大数 N，没有有效的方法能够将其分解，从而在已知（N，

D）的情况下无法获得 E；同样在已知（N，E）的情况下无法求得 D。

在本节前面演示 RSA 加密算法时，P、Q 的取值很小，使得 N 的值也很小，当得知（N，E）可以很容易地被破解计算出 D，根本不能保障安全性！

这里选择小的 P、Q 是因为通过手工计算，太大的数没办法处理，而在实际应用中必须选择较大的 P、Q，使得计算出的 N 足够大，这样不容易被破解。当然，随着计算机运算速度的提高，被破解的可能性也就变大了。现在小于 1024 位的 N 已经被证明是不安全的，因此不应使用小于 1024 位的 RSA，最好是使用 2048 位的 N。

例如，下面是一个 1024 位的 N 值及 D 和 E 值：

```
N=0x328C74784DF31119C526D18098EBEBB943B0032B599CEE13CC2BCE7B5FCD15F90B
    66EC3A85F5005DBDCDED9BDFCB3C4C265AF164AD55884D8278F791C7A6BFDAD5
    5EDBC4F017F9CCF1538D4C2013433B383B47D80EC74B5127CA05B5D6346B9EE5A
    D2D7BE7ABFB36E37108DD60438941D2ED173CCA50E114705D7E2BC511951
D=0x10001
E=0xE760A3804ACDE1E8E3D7DC0197F9CEF6282EF552E8CEBBB7434B01CB19A9D87
    A3106DD28C523C29954C5D86B36E943080E4919CA8CE08718C3B093086 7A98F63
    5EB9EA9200B25906D91B80A47B77324E66AFF2C4D70D8B1C69C50A9D8B4B7A3C9E
    E05FFF3A16AFC023731D806347 63DA1DCABE9861A4789BD782A592D2B1965
```

怎么样，看着这么长的 N 值头痛了吧（1024 比特二进制位是 256 位 16 进制数）。这么长的数不要说用手工计算，就是用计算机来计算，也需要专门编写处理大整数运算的相关方法，才能进行处理。

当将 N 值取为 2048 比特二进制位时，则计算量将更大，从而就可保障密文的安全。当计算机速度更快，能较快破解密文时，还需要将 N 值取得更大。

2.4　哥德巴赫猜想

说到素数的相关知识，就离不开哥德巴赫猜想。本节简单介绍哥德巴赫猜想的相关内容，了解什么是哥德巴赫猜想，怎么进行哥德巴赫猜想验证。

2.4.1　哥德巴赫猜想是什么

1742 年，在给欧拉的信中，哥德巴赫提出了以下猜想：

任一大于2的整数都可写成三个质数之和。

因现今数学界已经不使用“1 也是素数”这个约定，当初猜想的现代陈述为：

任一大于5的整数都可写成三个质数之和。

欧拉在回信中也提出另一等价版本，即：

任一大于2的偶数都可写成两个质数之和。

今日常见的猜想陈述为欧拉的版本，把命题：

任一充分大的偶数，都可以表示成为一个素因子个数不超过a个的数与另一个素因子个数不超过b个的数之和，记作“a+b”。

1966 年，我国的陈景润证明了“1+2”的成立，即：

任一充分大的偶数都可以表示成二个素数的和，或是一个素数和一个半素数的和。

对于哥德巴赫猜想的实际验证表明，至少 $4\sim10^{14}$ 以下的偶数都能表示成两个素数之和。很多时候，偶数表示成两个素数和的方法还不止一种，例如：

$$
\begin{aligned}
18&=5+13\\
&=7+11\\
64&=3+61\\
&=5+59\\
&=11+53\\
&=17+47\\
&=23+41
\end{aligned}
$$

大数学家欧拉相信这个猜想是正确的，但他不能证明。

叙述如此简单的问题，连欧拉这样首屈一指的数学家都不能证明，这个猜想便引起了许多数学家的注意。从哥德巴赫提出这个猜想至今，许多数学家都不断努力想攻克它，但都没有成功。

为什么说没有成功呢？由于在数列中，某一个数是否为素数是没有规律的，只能逐个推算。而对于一个很大的整数，要求出其等于两个素数之和，其计算量是非常巨大的，例如，要求 1 万位或 10 万位的整数是否等于两个素数之和，不要说手工计算，就是用计算机来推算，其程序设计的工作量都显得很复杂，需要处理大量的数据存储和转换过程。当然，对于一些比较小的整数，则可以很容易地求出来其等于两个素数之和。

2.4.2　数值验证

与不少数学猜想一样，数值上的验证也是哥德巴赫猜想的重要一环。

1938 年，尼尔斯 • 皮平（Nils Pipping）验证了所有小于 10^5 的偶数。

1964 年，M • L • 斯坦恩和 P • R • 斯坦恩验证了小于 10^7 的偶数。

1989 年，A • 格兰维尔将验证范围扩大到 2×10^{10}。

1993 年，Matti K. Sinisalo 验证了 4×10^{11} 以内的偶数。

2000 年，Jorg Richstein 验证了 4×10^{14} 以内的偶数。

至 2012 年 2 月为止，数学家已经验证了 3.5×10^{18} 以内的偶数，在所有的验证中，没有发现偶数哥德巴赫猜想的反例。

虽然不是数学家，但身为程序员的我们也可进行一翻哥德巴赫猜想的验证，例如：

首先检验：6=3+3；

接着检验：8=3+5；

接着检验：10=5+5；

……　……

下面利用计算机的快速计算能力，编写程序验证哥德巴赫猜想。

对于哥德巴赫猜想的验证，算法很简单，其基本思路是：设 N 为大于等于 6 的一个偶数，可将其分解为 N_1 和 N_2 两个数，分别检查 N_1 和 N_2 是否为素数，如都是，则在该数中得到验证。若 N_1 不是素数，就不必再检查 N_2 是否素数。先从 N_1=2 开始，检验 N_1 和 N_2（N_2=N-N_1）是否素数。然后是 N_1+2，再检验 N_1、N_2 是否素数……，直到 N_1=N/2 为止。

为了提高程序的效率，可对程序进行优化，首先根据 Eratosthenes 算法将指定范围中的素数都筛选出来保存到一个数组中，接着对整数 N 进行分解和判断。

根据上面的思路，编写相应的 C 程序如下：

```
#include <stdio.h>
#include <stdlib.h>
int CreatePrime(int n,int prime[])
{
   int i,j;
   for(i=2;i<=n;i++)                          //初始化数组
      prime[i]=1;                             //标志为1表示对应的数是质数
   prime[0]=prime[1]=0;
   for(i=2;i*i<=n;i++)                        //循环处理前 i 个
```

```
    {
        if(prime[i]==1)                                  //若为素数标记
        {
            for(j=2*i;j<=n;j++)                          //筛去合数
            {
                if(j%i==0)                               //能被整除
                    prime[j]=0;                          //不是素数
            }
        }
    }
}
int main()
{
    int n,i,j,flag;
    int *prime;
    printf("哥德巴赫猜想验证\n");
    printf("输入一个要验证的最大数 n(n>=6):");
    scanf("%d",&n);
    if(n<6)                                              //判断输入数据是否合法
    {
        printf("数据输入错误!\n");
        return 0;
    }
    if(!(prime=(int *)malloc(sizeof(int)*n)))
    {
        printf("分配内存失败!\n");
        getch();
        return 0;
    }
    CreatePrime(n,prime);                                //生成素数数组
    for(i=6;i<=n;i+=2)                                   //从 6 开始，循环验证各偶数
    {
        flag=1;
        for(j=2;j<=i/2;j++)                              //判断组成每个数的两个加数
        {
          if(j%2==0 || ((i-j)%2==0)) continue; //若一个加数是偶数，不判断素数
          if(prime[j]==1 && prime[i-j]==1)               //若两个加数都是素数
          {
              printf("%d=%d+%d\n",i,j,i-j);              //输出素数
              flag=0;                                    //清除标志
              break;
          }
        }
        if(1==flag)                              //若某个偶数不是由两个奇数组成
```

```
            printf("找到一个不符合要求的偶数:%d\n",i);
    }
    getch();
    return 0;
}
```

执行以上程序，输入 1000000（一百万），验证 1000000 以内的偶数，运行结果如图 2-12 所示。

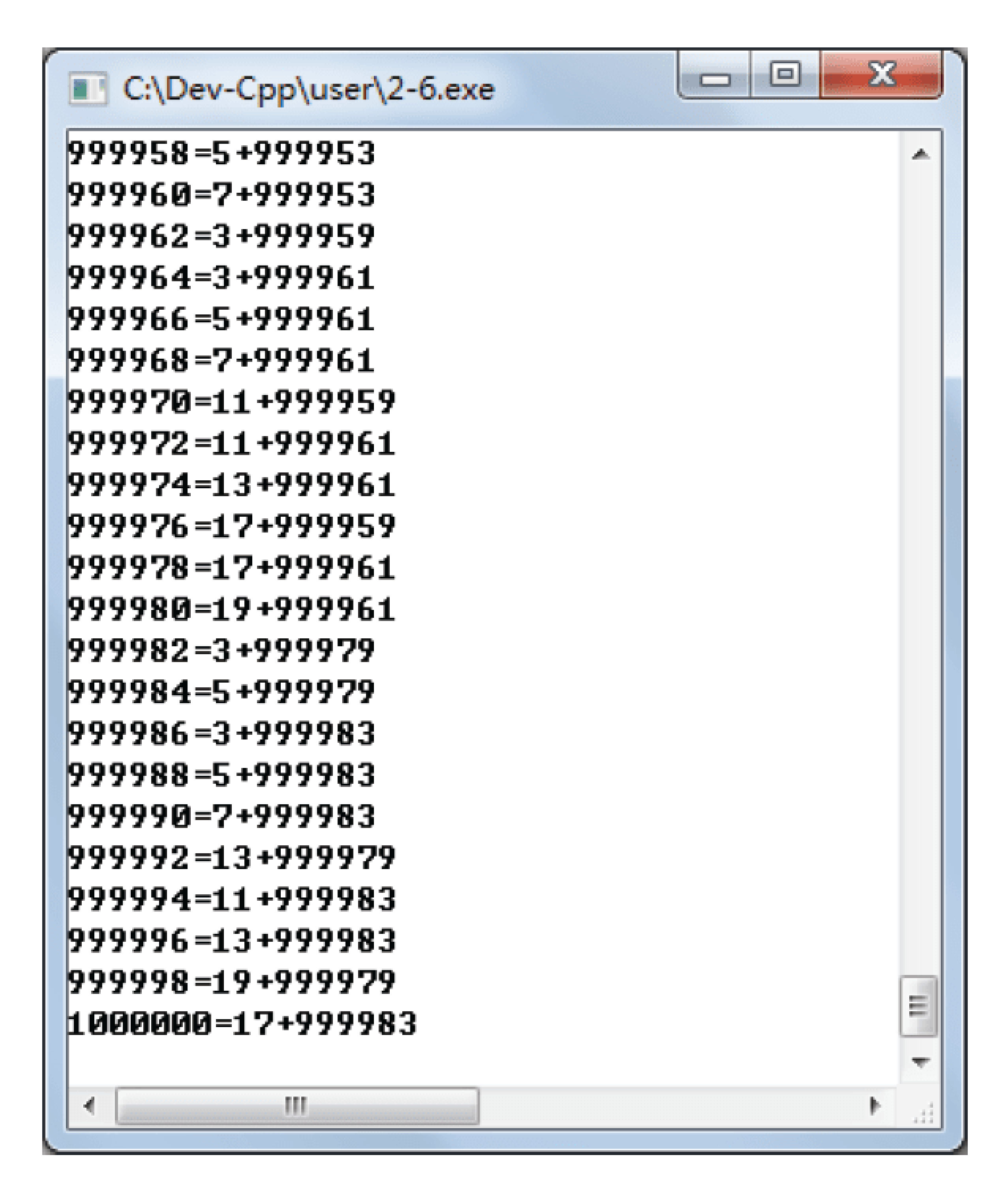


图 2-12

从图 2-12 中的运行结果可看出，在 1000000 以内的所有偶数都通过了验证。

2.5　梅 森 素 数

我们知道，在自然数中素数较少，10000 以内的自然数中只有 1229 个素数，而孪生素数就更少，10000 以内的孪生素数只有 205 对（510 个）。在素数中还有更稀少的一个种类，那就是梅森素数，截止 2013 年 2 月，人们仅发现 48 个梅森素数。

2.5.1　什么是梅森素数

那么，什么是梅森素数呢？

马林•梅森是 17 世纪法国著名的数学家和修道士，也是当时欧洲科学界一位独特的中心人物。梅森在欧几里得、费马等人的有关研究的基础上对 $2^P–1$ 作了大量的计算、验证工作，并于 1644 年在他的《物理数学随感》一书中断言：对于 P=2、3、5、7、13、17、19、31、67、127、257 时，$2^P–1$ 是素数；而对于其他所有小于 257 的数时，$2^P–1$ 是合数。前面的 7 个数（即 2、3、5、7、13、17 和 19）属于被证实的部分，是他整理前人的工作得到的；而后面的 4 个数（即 31、67、127 和 257）属于被猜测的部分。不过，人们对其断言仍深信不疑，连大数学家莱布尼兹和哥德巴赫都认为它是对的。

梅森的猜测中遗漏了 61、89、107，而将 67 和 257 被错误地包含了进来。

虽然梅森的断言中包含错误，但他的工作极大地激发了人们研究 $2^P–1$ 型素数的热情，使其摆脱作为“完美数”的附庸的地位。由于梅森最早系统而深入地研究 $2^P–1$ 型的数，为了纪念他，数学界就把这种数称为“梅森数”；并以 M_p 来标记梅森数，即 $M_p=2^P–1$。如果梅森数为素数，则称之为“梅森素数”（即 $2^P–1$ 型素数）。

总结一下，梅森素数是指形如 $2^P–1$ 的正整数，其中指数 P 是素数，常记为 M_P。若 M_P 是素数，则称为梅森素数。

当 P=2、3、5、7 时，M_P 都是素数，但 P=11 时，M_{11} 不是素数，因此 M_{11} 不是梅森素数，如图 2-13 所示。

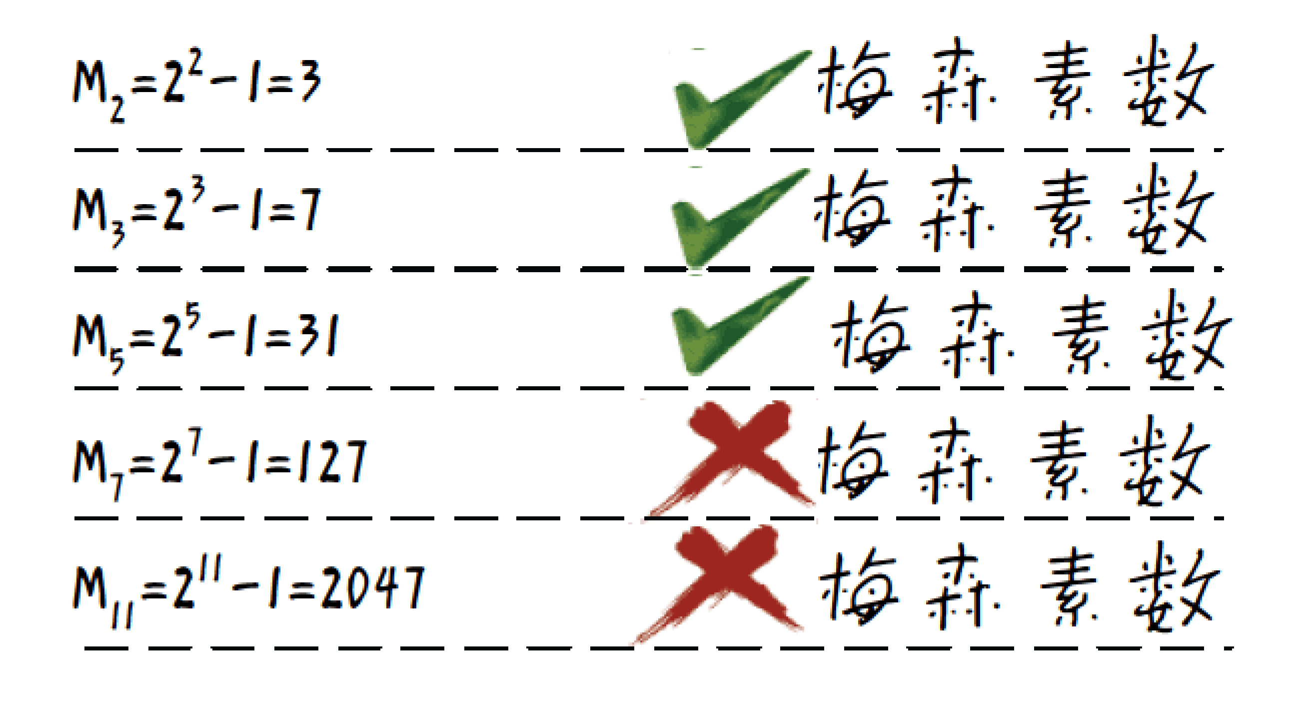


图 2-13

2.5.2 已知的梅森素数列表

是否有无穷多个梅森素数？这是数论中未解决的难题之一。截止 2013 年 2 月累计发现 48 个梅森素数，最大的是 P=57885161（即 $M_{57885161}=2^{57885161}–1$），此时 M_P 是一个 17,425,170 位的非常大的整数。

如表 2-1 所示是目前为止已发现的梅森素数列表。

表 2-1 梅森素数列表

序号	P	M_P	M_P的位数	发现日期	发现者
1	2	3	1	公元前 5 世纪	古希腊数学家
2	3	7	1	公元前 5 世纪	古希腊数学家
3	5	31	2	公元前 3 世纪	古希腊数学家

续表

序号	P	Mp	Mp 的位数	发现日期	发现者
4	7	127	3	公元前 3 世纪	古希腊数学家
5	13	8191	4	1456 年	无名氏
6	17	131071	6	1588 年	Cataldi
7	19	524287	6	1588 年	Cataldi
8	31	2147483647	10	1772 年	欧拉
9	61	2305843009213693951	19	1883 年	Pervushin
10	89	618970019…449562111	27	1911 年	Powers
11	107	162259276…010288127	33	1914 年	Powers
12	127	170141183…884105727	39	1876 年	卢卡斯
13	521	686479766…115057151	157	1952 年 1 月 30 日	Robinson
14	607	531137992…031728127	183	1952 年 1 月 30 日	Robinson
15	1,279	104079321…168729087	386	1952 年 6 月 25 日	Robinson
16	2,203	147597991…697771007	664	1952 年 10 月 7 日	Robinson
17	2,281	446087557…132836351	687	1952 年 10 月 9 日	Robinson
18	3,217	259117086…909315071	969	1957 年 9 月 8 日	Riesel
19	4,253	190797007…350484991	1,281	1961 年 11 月 3 日	Hurwitz
20	4,423	285542542…608580607	1,332	1961 年 11 月 3 日	Hurwitz
21	9,689	478220278…225754111	2,917	1963 年 5 月 11 日	Gillies
22	9,941	346088282…789463551	2,993	1963 年 5 月 16 日	Gillies
23	11,213	281411201…696392191	3,376	1963 年 6 月 2 日	Gillies
24	19,937	431542479…968041471	6,002	1971 年 3 月 4 日	布莱恩特・塔克曼
25	21,701	448679166…511882751	6,533	1978 年 10 月 30 日	Noll & Nickel
26	23,209	402874115…779264511	6,987	1979 年 2 月 9 日	Noll
27	44,497	854509824…011228671	13,395	1979 年 4 月 8 日	Nelson & Slowinski
28	86,243	536927995…433438207	25,962	1982 年 9 月 25 日	Slowinski
29	110,503	521928313…465515007	33,265	1988 年 1 月 28 日	Colquitt & Welsh
30	132,049	512740276…730061311	39,751	1983 年 9 月 20 日	Slowinski
31	216,091	746093103…815528447	65,050	1985 年 9 月 6 日	Slowinski
32	756,839	174135906…544677887	227,832	1992 年 2 月 19 日	Slowinski & Gage
33	859,433	129498125…500142591	258,716	1994 年 1 月 10 日	Slowinski & Gage
34	1,257,787	412245773…089366527	378,632	1996 年 9 月 3 日	Slowinski & Gage
35	1,398,269	814717564…451315711	420,921	1996 年 11 月 13 日	GIMPS / Joel Armengaud
36	2,976,221	623340076…729201151	895,932	1997 年 8 月 24 日	GIMPS / Gordon Spence
37	3,021,377	127411683…024694271	909,526	1998 年 1 月 27 日	GIMPS / Roland Clarkson
38	6,972,593	437075744…924193791	2,098,960	1999 年 6 月 1 日	GIMPS / Nayan Hajratwala
39	13,466,917	924947738…256259071	4,053,946	2001 年 11 月 14 日	GIMPS / Michael Cameron
40	20,996,011	125976895…855682047	6,320,430	2003 年 11 月 17 日	GIMPS / Michael Shafer
41	24,036,583	299410429…733969407	7,235,733	2004 年 5 月 15 日	GIMPS / Josh Findley
42	25,964,951	122164630…577077247	7,816,230	2005 年 2 月 18 日	GIMPS / Martin Nowak
43*	30,402,457	315416475…652943871	9,152,052	2005 年 12 月 15 日	GIMPS / Curtis Cooper 及 Steven Boone

续表

序号	P	M_P	M_P 的位数	发现日期	发现者
44*	32,582,657	124575026…053967871	9,808,358	2006 年 9 月 4 日	GIMPS / Curtis Cooper 及 Steven Boone
45*	37,156,667	202254406…308220927	11,185,272	2008 年 9 月 6 日	GIMPS / Hans-Michael Elvenich
46*	42,643,801	169873516…562314751	12,837,064	2009 年 4 月 12 日	GIMPS / Odd M. Strindmo
47*	43,112,609	316470269…697152511	12,978,189	2008 年 8 月 23 日	GIMPS / Edson Smith
48*	57,885,161	581887266…724285951	17,425,170	2013 年 1 月 25 日	GIMPS / Curtis Cooper

由于现在还不知道在第 42 个梅森素数（$M_{25,964,951}$）至第 48 个梅森素数（$M_{57,885,161}$）之间是否还存在未知梅森素数，所以在其序号之前用*标出。

第 3 章　递归——自己调用自己

递归（Recursion），又译为递回，在数学与计算机科学中，是指在函数的定义中又调用函数自身的方法。递归是一种奇妙的思考问题的方法，通过递归的这种思路，可简化问题的定义。

3.1　从前有座山，山里有座庙

递归一词常用于描述以自相似方法重复事物的过程。例如，当两面镜子相互之间近似平行时，镜中嵌套的图像是以无限递归的形式出现的。

3.1.1　老和尚讲的故事

将时光往前推，在中国还流传着这样一个有趣的故事：

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？‘从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……’”

怎么样？这个故事就是不断重复着老和尚讲的故事，有趣？还是无聊？

不管怎样，这个故事可以不断地重复，一直讲下去。

这就是生活中一个用递归形成的故事。

3.1.2　德罗斯特效应

再来看一张图，如图 3-1 所示为一张网页的截图，在网页图中又包括相同的一份较小的截图，在小图中又包含一份更小的截图，……，这样就形成了一幅递归形式的图形。

图 3-1 所示的图形称为德罗斯特效应（Droste effect），是递归的一种视觉形式，是指一张图片的某个部分与整张图片相同，如此产生无限循环。这种照片是通过名为 Mathmap 的数学软件制作出来，使用 PhotoShop 的 Droste Effect 滤镜也可制作出这种效果。

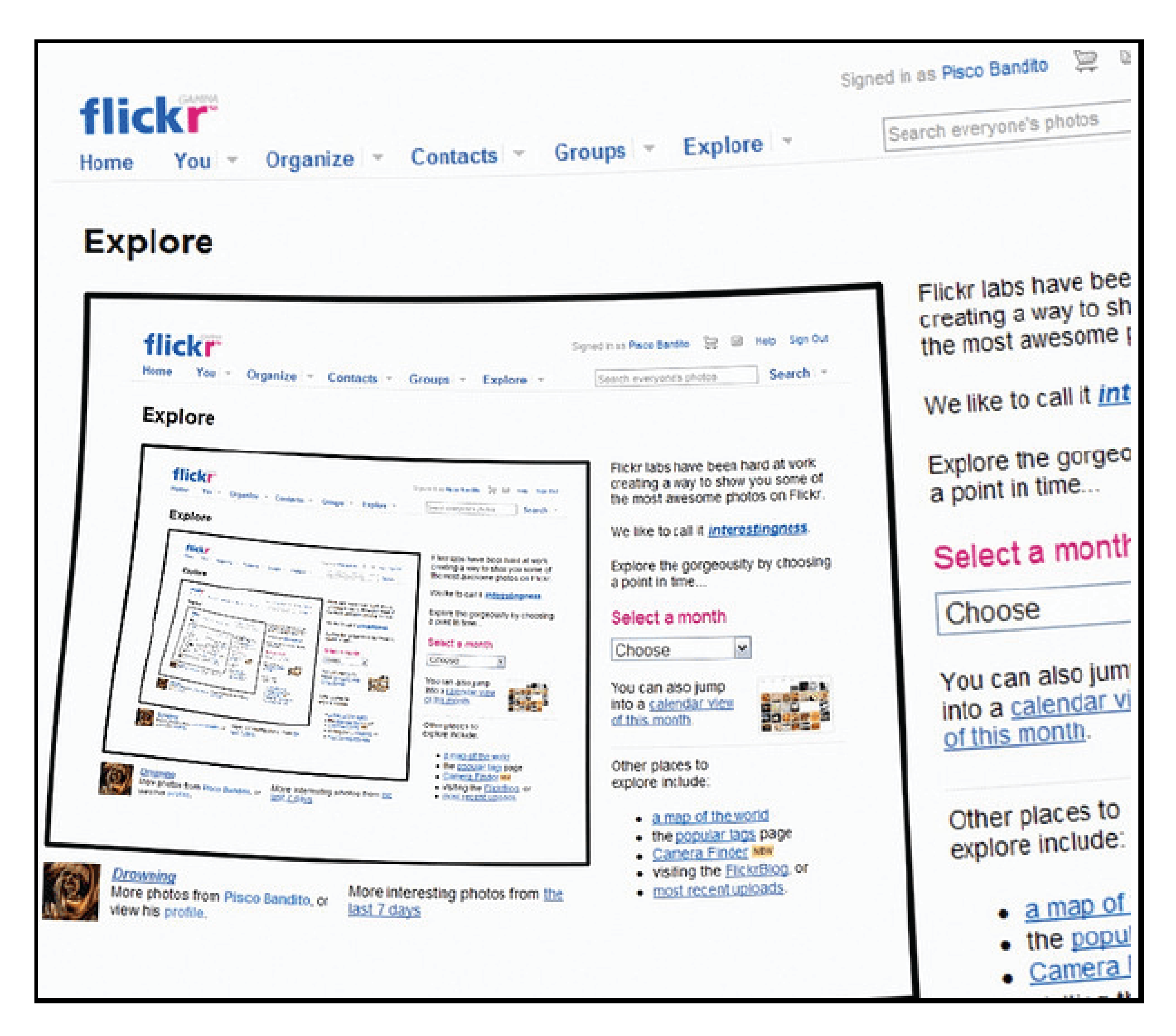


图 3-1

3.1.3 什么是递归

那么，什么是递归呢？

在数学和计算机科学中，递归指由一种（或多种）简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况。

例如，我们人类的发展繁衍中，人之间的辈份就是一种递归（如图 3-2 所示），在这个递归中首先定义一个基本情况，接着递归定义，具体情况如下：

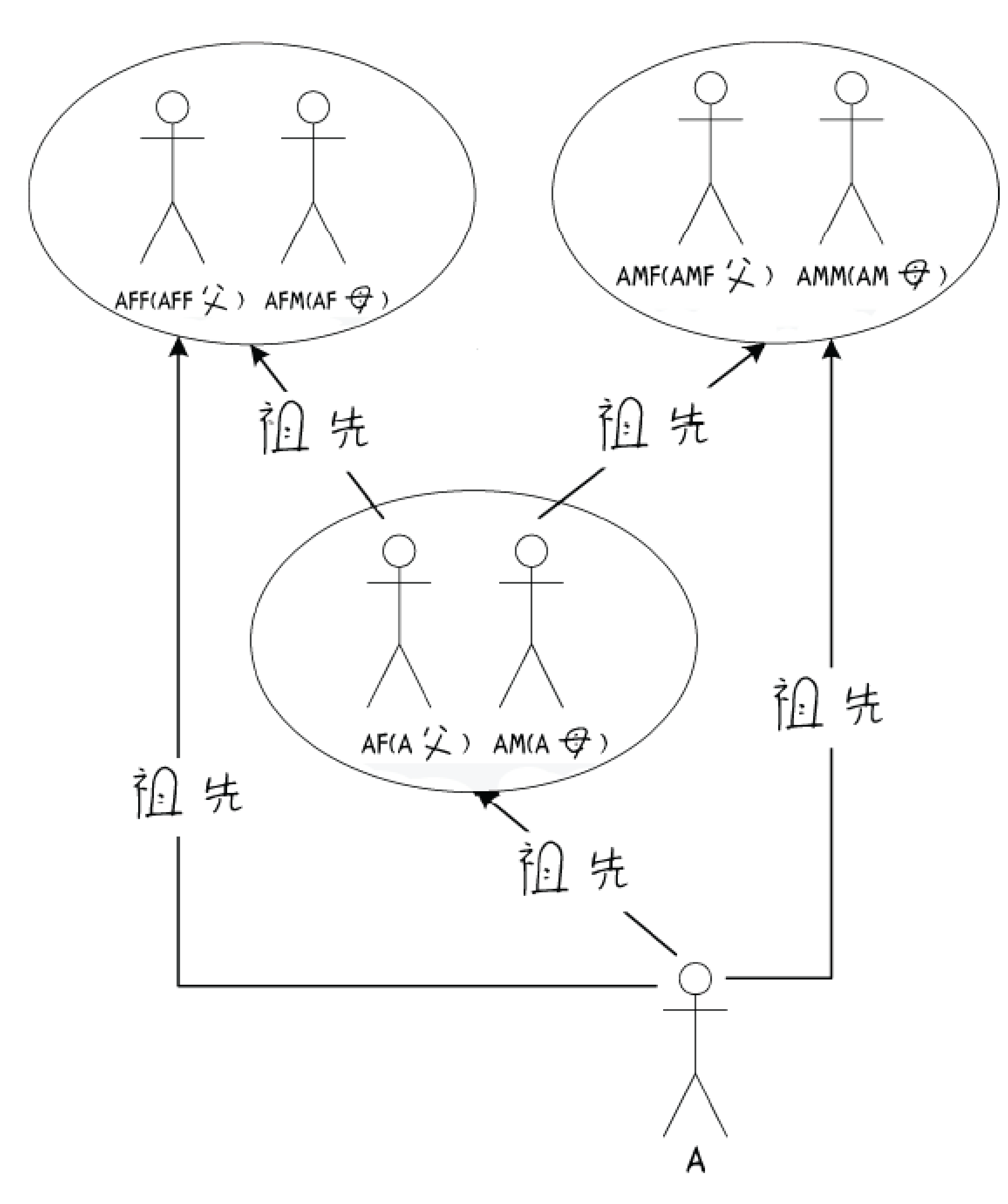


图 3-2

- A 的父母是 A 的祖先（这是基本情况）。
- A 祖先的双亲同样是 A 的祖先（递归步骤）。

对于递归，一种便于理解的心理模型，认为递归对对象的定义是按照“先前定义的”同类对象来定义的。

例如：要求能移动 100 个箱子，该怎么移动？

首先移动 1 个箱子，并记下它移动到的位置，然后再去解决较小的问题（由于已移动了 1 个箱子，剩下 99 个箱子），这时的问题就简化为“怎样才能移动 99 个箱子？”，……不断重复，到最后，问题将简化为“怎样移动 1 个箱子”，如图 3-3 所示。

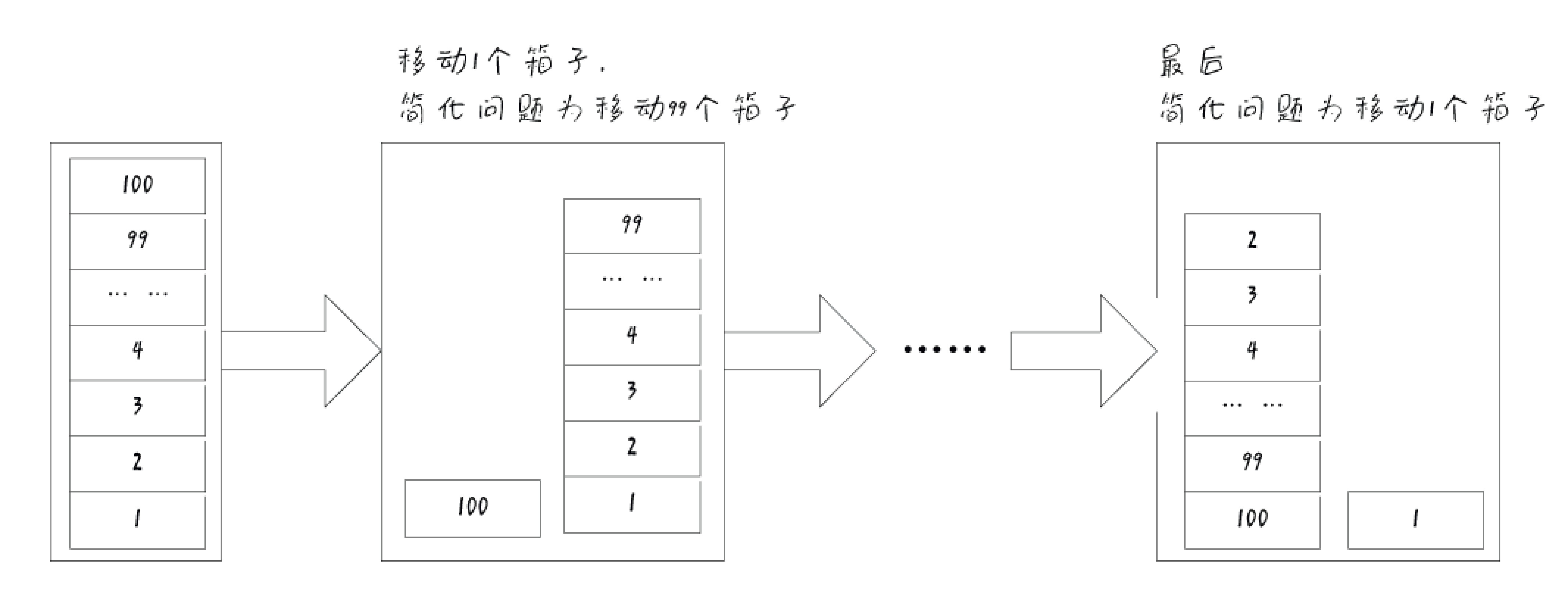


图 3-3

类似的定义在数学中十分常见。例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数，如图 3-4 所示。

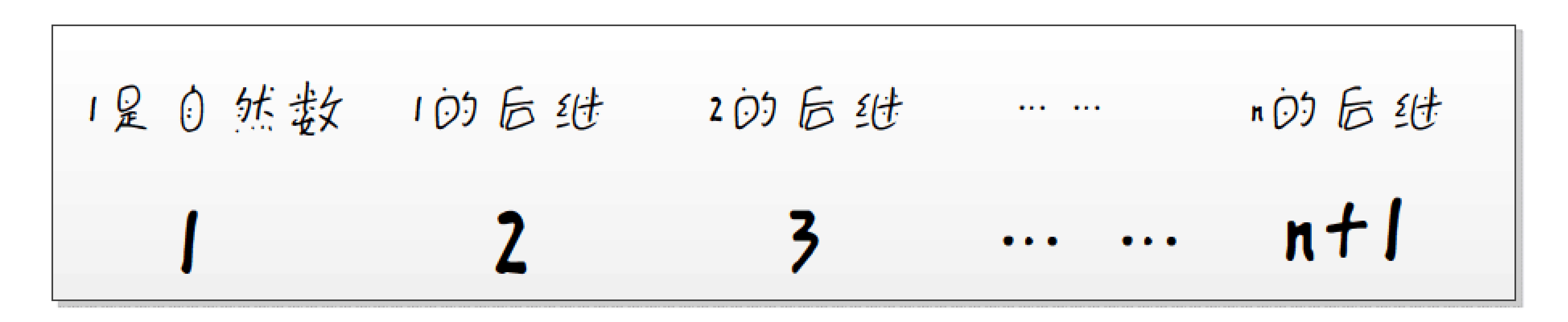


图 3-4

3.1.4　用递归能解决哪些问题

递归是一种非常接近自然思维的思想，其实了解多了以后，用起递归来是非常自然的，但不是每个场合使用递归都是合适的。通常递归方法适合于层次结构本身就是递归定义的情况，比如二叉树的遍历，因为二叉树的定义就是“一颗空树，或者一个节点+左右两颗子二叉树”，它的定义就是递归的，所以用递归操作相当方便。

简单来说，递归问题，可以划分为一个或多个子问题，而处理子问题的规则与处理原问题的规则是一样的。

如图 3-2 所示的祖先问题、用集合论对自然数的定义问题，以及老和尚讲的故事，都是无穷无尽的。通常意义上来说，对于一个无穷尽的问题，是没办法编写程序来解决的，因为程序没有办法结束。

虽然常见的递归问题都没有尽头，不过，我们可以找到起点，这时可从某一个指定的终点开始，向起点方向运行，当到达起点位置时，即可结束递归调用。

那么，在实际应用中要使用递归算法，通常需要分析以下 3 方面的问题：

- 每一次递归调用，在处理问题的规模上都应有所缩小（通常问题规模可减半）。
- 相邻两次递归调用之间有紧密的联系，前一次要为后一次递归调用做准备，通常是前一次递归调用的输出作为后一次递归调用的输入。
- 在问题的规模极小时，必须直接给出解答而不再进行递归调用，因而每次递归调用都是有条件的（以规模未达到直接解答的大小为条件），无条件递归调用将会成为死循环而不能正常结束。

根据上面的描述，在设计递归算法时，主要需考虑以下两方面的问题：

- 确定递归公式。把规模大的、较难解决的问题变成规模较小、易解决的同一问题，需要通过哪些步骤或等式来实现？这是解决递归问题的难点。
- 确定边界（终了）条件。在什么情况下可以直接得出问题的解？这就是问题的边界条件及边界值。

3.1.5　一个简单例子：求最大公约数

求最大公约数是数学中一个简单的问题。

如果有一个自然数 A 能被自然数 B 整除，则称 A 为 B 的倍数，B 为 A 的约数。几个自然数公有的约数，叫做这几个自然数的公约数。这些公约数中最大的一个公约数，称为这几个自然数的最大公约数，简称为 GCD。

例如，在 2、4、6 这 3 个数中，2 就是 2、4、6 的最大公约数。

怎么求最大公约数呢？早在公元前 300 年左右，欧几里得就在他的著作《几何原本》中给出了高效的解法——辗转相除法。

辗转相除法的方法是用较大的数 M 除以较小的数 N，较小的除数 N 和得出的余数 R 构成新的一对数，继续重复前面的除法（用较大数除以较小数），直到出现能够整除的两个数，其中较小的数（即除数）就是最大公约数。

例如，求 153 和 123 的最大公约数，操作过程如下：

$$153 \div 123 = 1 \cdots\cdots 30$$

$$123 \div 30 = 4 \cdots\cdots 3$$

$$30 \div 3 = 10 \cdots\cdots 0$$

所以，153 和 123 的最大公约数就是 3。

可以看出，在用辗转相除法求最大公约数时，每次相除后的除数 N 和余数 R 都将原来的问题规模缩小，而且有一个边界条件（两数能够整除，即余数为 0）。有这两条，就可使用递归算法来解这个问题，处理流程如图 3-5 所示。

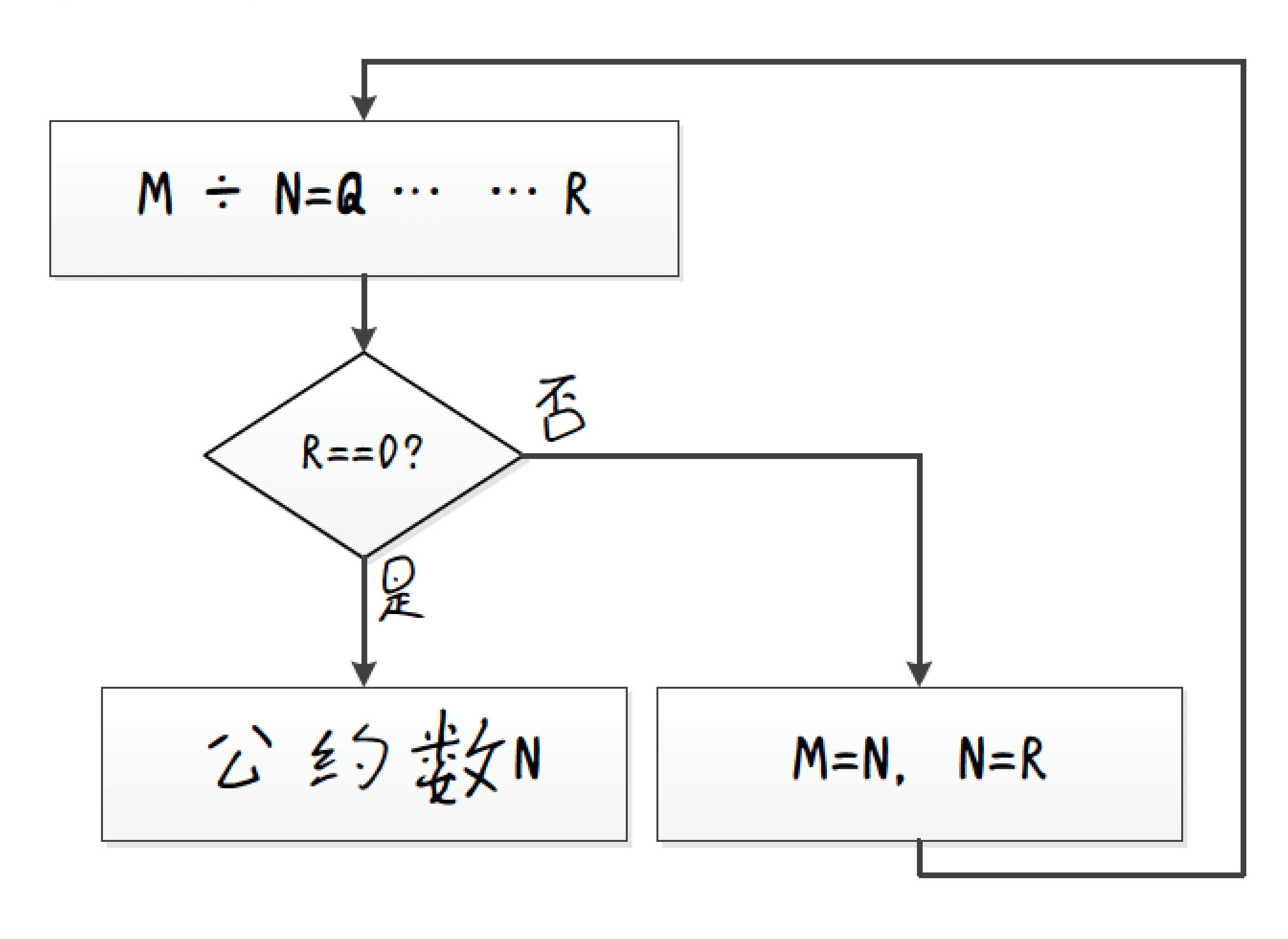


图 3-5

根据图 3-5 所示的流程，可编写出以下 C 语言程序。

```
#include <stdio.h>

int gcd(int m,int n)
{
    int r;
    r=m%n;
    if(r==0)
    {
        return n;
    }else{
        return gcd(n,r);
    }
}

int main()
{
    int m,n;
    printf("请输入 2 个整数: ");
    scanf("%d,%d",&m,&n);
    printf("%d,%d 的最大公约数为: %d\n",m,n,gcd(m,n));
    getch();
    return 0;
}
```

在以上程序中，定义了一个 gcd()函数，在这个函数中调用 gcd()函数，这样就形成了递归调用。在这个函数内部通过辗转相除，然后判断余数是否为 0，若为 0 就返回 *n* 值，否则递归调用 gcd()函数进行辗转相除。

怎么样，递归调用函数是不是很简单呢？

3.2 用递归计算阶乘

阶乘（factorial）是基斯顿・卡曼于 1808 年发明的运算符号。阶乘也是数学里的一种术语，在很多计算中都要使用到阶乘。

3.2.1 阶乘该怎么计算

一个正整数的阶乘是所有小于或等于该数的正整数的积，并且有 0 的阶乘为 1 的约定。自然数 *n* 的阶乘写作 *n*!。

例如，求 5 的阶乘的算式如下：

$$5!=1\times 2\times 3\times 4\times 5$$
$$=120$$

从上式可看出，当求 *n* 的阶乘时，就从 1 逐项相乘到 *n* 即可，这是一个循环结构，因此编写一个循环程序就可很容易地求出数据 *n* 的阶乘，在第 1 章中曾给出过一个通过循环计算阶乘的 C 语言程序。

由于阶乘的结果会呈几何级数增长，在 32 位计算机中，只能计算到 13 的阶乘，14 的阶乘结果就超过 32 位二进制的表示。即使是在 64 位字长的计算机中，能表示的数据大小也是有限的，只能保存 20 的阶乘。

如果需要求很大的数的阶乘（例如求 1000 的阶乘），就不能简单地使用第 1 章中介绍的阶乘程序来进行运算了。

如何超越计算机变量的取值范围来计算阶乘？这就需要考虑编写出能处理大整数的函数（在 C# 4 中已要提供了大整数功能），才能计算更大数的阶乘。由于篇幅所限，这里不单独编写大整数的函数，而是提供另外一种相对简单的求大数阶乘的方法。

这种思路就是：考虑将多位数相乘化解为一位数相乘，例如，11 的阶乘为 39916800，若需要求 12 的阶乘，则需要将 39916800 与 12 相乘，按手工计算乘法的竖式方法，可用 2 与 39916800 相乘的结果加上用 1 与 39916800 相乘的结果，然后再将结果相加，得到 12 的阶乘，如图 3-6 左图所示。由于前一数的阶乘的结果很大，按左图的方式计算也很容易导致溢出。根据乘法交换律，可以将左图所示算式转换为右图所示算式，这样，每次计算的结果就不容易出现溢出。

按图 3-6 所示的思路，定义一个数组，使数组的每一个数组元素保存阶乘的一位结果（如图 3-6 右图所示中，11 的阶乘结果为 8 位，就用数组中的 8 个数组元素分别保存这 8 位）。当要计算 12 的阶乘时，可按以下步骤进行计算。

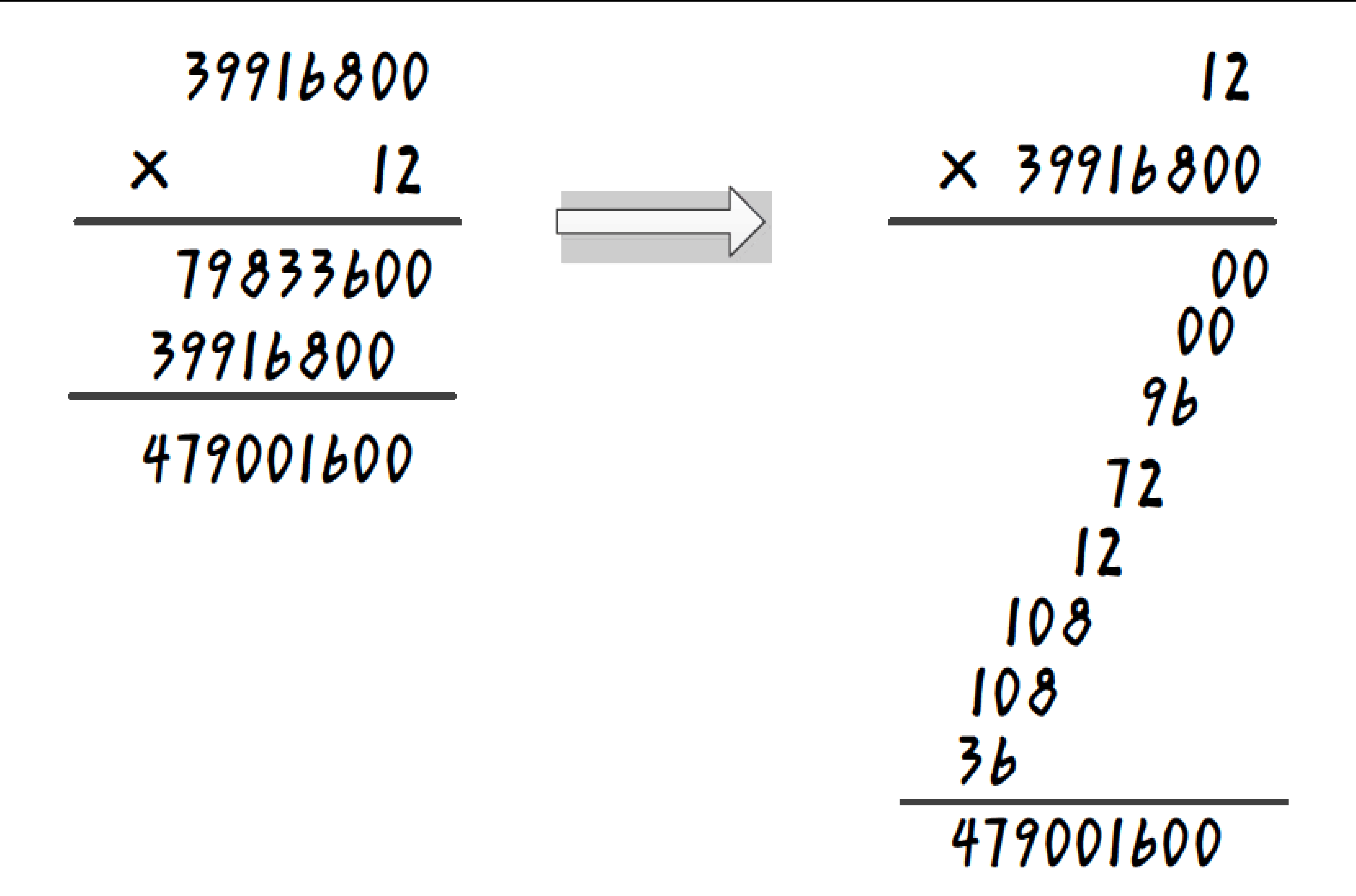


图 3-6　乘法竖式

（1）用 12 去乘以数组中的每个元素，并将结果保存到原来的数组元素中。

（2）判断每个数组元素中的值是否大于 9，若大于 9 则进行进位操作。通过进位操作，使数组中每个元素保存的值都只有一位数。

具体操作过程如图 3-7 所示。

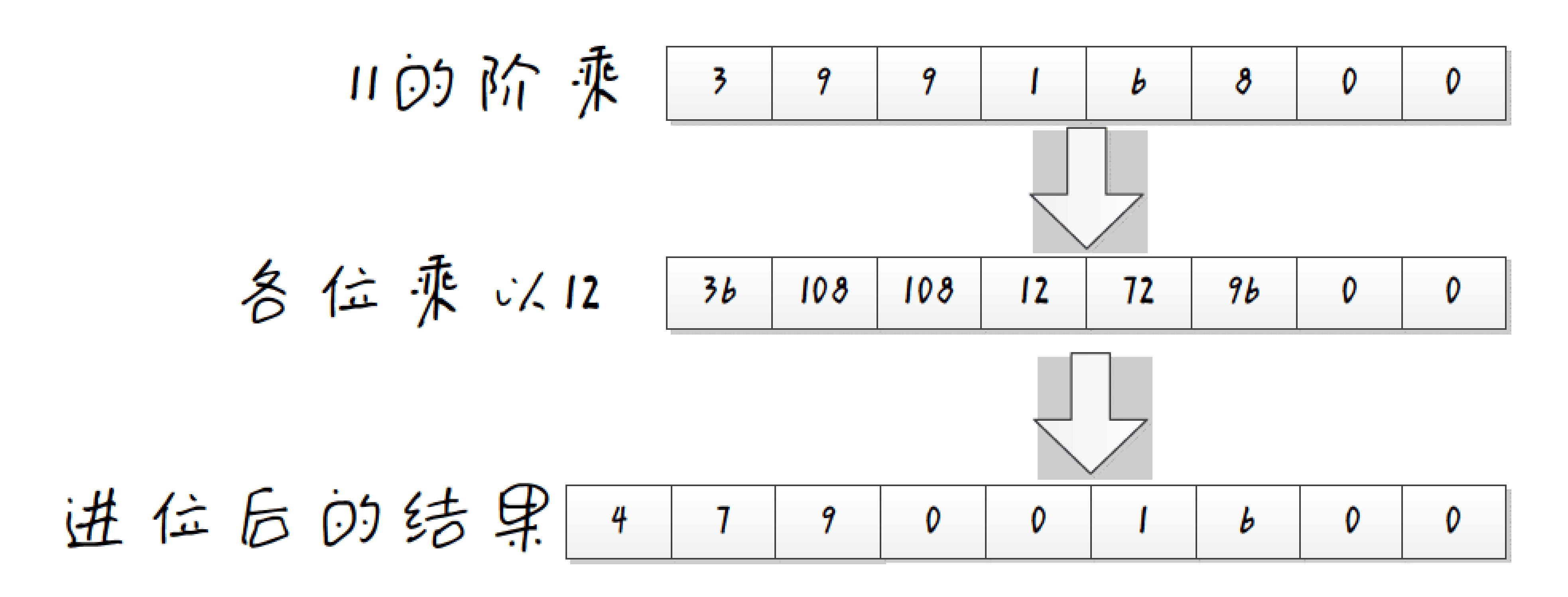


图 3-7

通过这种方式，就可以计算出计算机整型变量所能表示数据的十分之一这么大的数的阶乘了（因为数组中保存的是 0～9 中的 1 位数，而为了使结果不超过整型变量表示范围，与数组中各元素相乘的数据只能是计算机整型变量所能表示数据的十分之一）。

按这种思路，编写能计算较大整数阶乘的程序，具体代码如下：

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void carry(int bit[],int pos)                //计算进位
{
```

```
    int i,carray=0;
    for(i=0;i<=pos;i++)                         //从 0~pos 逐位检查是否需要进位
    {
        bit[i]+=carray;                         //累加进位
        if(bit[i]<=9)                           //小于 9 不进位
            carray=0;
        else if(bit[i]>9 && i<pos)              //大于 9 但不是最高位
        {
            carray=bit[i]/10;                   //保存进位值
            bit[i]=bit[i]%10;                   //得到该位的一位数
        }
        else if(bit[i]>9 && i>=pos)             //大于 9 且是最高位
        {
            while(bit[i]>9)                     //循环向前进位
            {
                carray=bit[i]/10;               //计算进位值
                bit[i]=bit[i]%10;               //当前的一位数
                i++;
                bit[i]=carray;                  //在下一位保存进位的值
            }
        }
    }
}

int main()
{
    int num,pos,digit,i,j,m,n;
    double sum=0;
    int *fact;                                      //保存阶乘结果的数组
    printf("输入计算其阶乘的数：Num=");
    scanf("%d",&num);
    for(i=1;i<=num;i++)                             //计算阶乘的位数
        sum+=log10(i);
    digit=(int)sum+1;                               //阶乘结果的数据长度
    if(!(fact=(int *)malloc((digit+1)*sizeof(int))))//分配保存阶乘位数的内存
    {
        printf("分配内存失败!\n");
        return 0;
    }

    for(i=0;i<=digit;i++)                           //初始化数组
        fact[i]=0;

    fact[0]=1; //设个位为 1

    for(i=2;i<=num;i++)                         //将 2~num 逐个与原来的积相乘
    {
        for(j=digit;j>=0;j--)                       //查找最高位
            if(fact[j]!=0)
```

```
            {
                pos=j;                                      //记录最高位
                break;
            }
        for(j=0;j<=pos;j++)
            fact[j] *=i;                                    //每一位与 i 乘
        carry(fact,pos);                                    //进位处理
    }

    for(j=digit;j>=0;j--)                                   //查找最高位
        if(fact[j]!=0)
        {
            pos=j;                                          //记录最高位
            break;
        }

    m=0;                                                    //统计输出位数
    n=0;                                                    //统计输出行数
    printf("\n 输出%d 的阶乘结果(按任意键显示下一屏):\n",num);
    m=2-pos%3;                                              //按每 3 位分组，计算第一组

    for(i=pos;i>=0;i--)                                     //输出计算结果
    {
        printf("%d",fact[i]);
        m++;
        if(m%3==0)                                          //每 3 个数字输出一个空格
            printf(" ");
        if(40==m)                                           //每行输出 40 个数字
        {
            printf("\n");
            m=0;
            n++;
            if(10==n)                                       //输出 10 行则暂停
            {
                getch();
                printf("\n");
                n=0;
            }
        }
    }

    printf("\n\n");
    printf("%d 的阶乘共有%d 位。\n",num,pos+1);
    getch();
    return 0;
}
```

执行以上程序计算 1000 的阶乘，可得到如图 3-8 所示的结果。从结果中可看到，1000 的阶乘共有 2568 位。

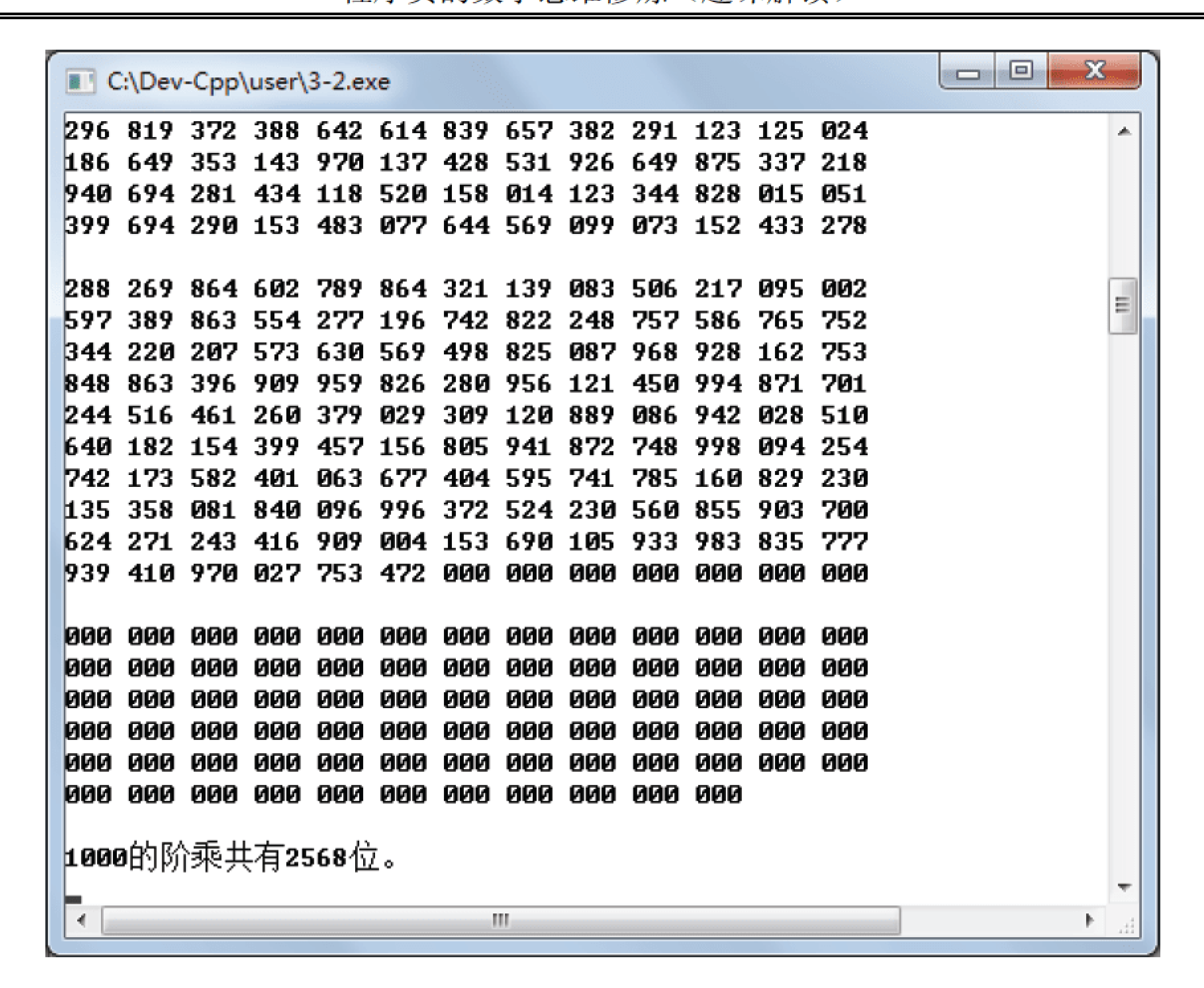


图 3-8

3.2.2 阶乘的递归计算方法

从前面的运算过程可看出，阶乘是一个规模比较大的问题，这时可考虑通过递归的方式来计算阶乘。

在定义递归算法时，需要考虑 3 个方面的问题。

首先，每一次递归调用，处理问题的规模应有所缩小，在阶乘中可以做到，如求 n 的阶乘，可将其分解为如下形式：

$$n! = n \times (n-1)!$$

⇩

$$(n-1)! = (n-1) \times (n-2)!$$

由上式可看到，n 的阶乘可分解为 n 乘以（n-1）的阶乘，而（n-1）的阶乘又可分解为（n-1）与（n-2）的阶乘。

这样，每次运算就可将问题的规模缩小。

其次，在定义递归算法时，相邻两次递归调用之间有紧密的联系，前一次为后一次递归调用做准备。在上式中可看到，n 的阶乘分解后，下一次递归调用时的输入就为（n-1）。

最后，在问题的规模极小时，必须直接给出解答而不再进行递归调用。在阶乘的递归算法中，计算到最后，求 0 的阶乘时，就不用再递归，直接返回其结果为 1 即可。

根据上面的分析，可用以下方式定义阶乘的递归算法。

$$n! = \begin{cases} 1 & (n=0) \\ n \times (n-1)! & (n>0) \end{cases}$$

根据以上定义，可用 C 语言编写出如下递归计算阶乘的程序。

```
#include <stdio.h>

int fact(int n)
{
    if(n==0)                                //递归的结束条件
    {
        return 1;
    }else{
        return n*fact(n-1);                 //递归调用
    }
}

int main()
{
    int n;
    printf("请输入要计算阶乘的整数：");
    scanf("%d",&n);

    printf("%d!=%d\n",n,fact(n));
    getch();
    return 0;
}
```

可以看出，这个程序比第 1 章中编写的阶乘程序简单。

3.2.3　递归的过程

要理解递归，首先应了解一种数据结构：堆栈（简称栈）的概念。

栈是一个后进先出的压入（push）和弹出（pop）式数据结构。在程序运行时，系统每次向栈中压入一个对象，然后栈指针向上移动一个位置。当系统从栈中弹出一个对象时，最近进栈的对象将被弹出，然后栈指针向下移动一个位置。

C 编译器处理函数调用时，就是使用栈来保存数据的。当主调函数调用另一个函数时，C 编译器将主调函数的所有实参和返回地址压入到栈中，栈指针将移到合适的位置来容纳这些数据。

当进行被调函数时，编译器将栈中的实参数据弹出，赋值给函数的形参。在被调用函数执行期间，还可利用栈来保存函数执行时的局部变量。当被调用函数准备返回时，系统将弹出栈中所有当前函数压入栈中的值，这时，栈指针移动到被调用函数刚开始执

行时的位置。接着被调用函数返回，系统从栈中弹出返回地址，主调函数就可以继续执行了。

如图 3-9 所示就是计算机中栈的示例，左图所示是“函数 1”调用“函数 2”的情况，在调用“函数 2”时，计算机将“函数 2”的返回地址压入栈中，接着将函数参数压入栈中。而右图显示“函数 2”调用结束返回“函数 1”后的情况，这时“函数 2”中的参数、返回地址都从栈中弹出。

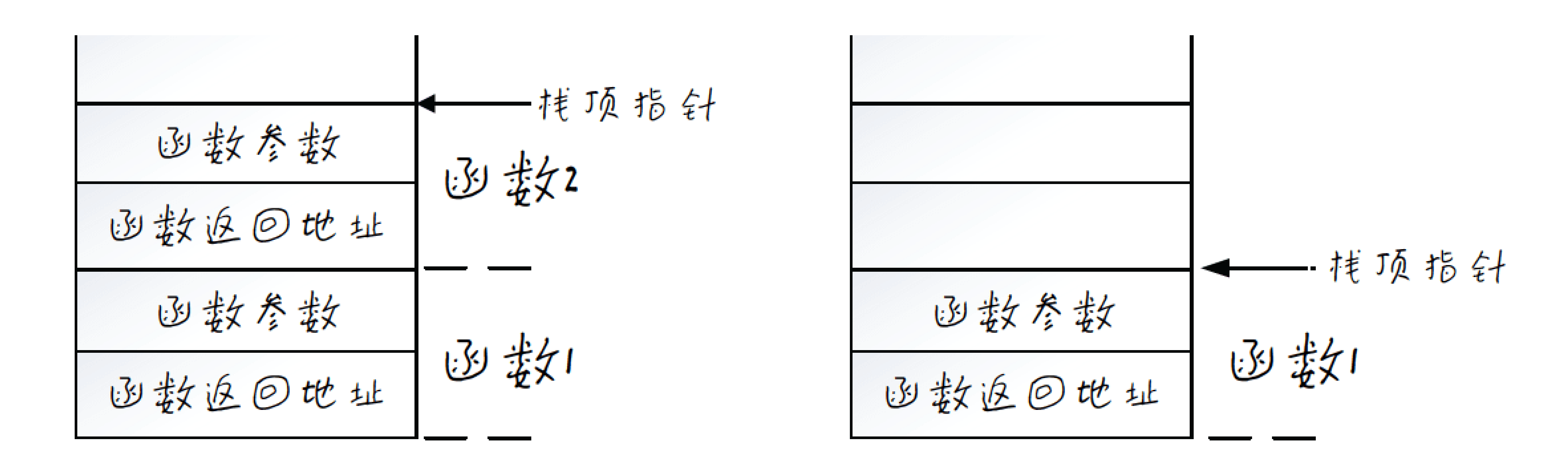


图 3-9

栈在每个程序中都是存在的，它不需要程序员编写代码去维护，而是由系统自动处理。

递归之所以能实现，是因为函数的每个执行过程都在栈中有自己的形参和局部变量的备份，这些备份和函数的其他执行过程毫不相干。这种机制是大多数程序设计语言实现子程序结构的基础，使得递归成为可能。

假定某个主调函数调用了一个被调用函数，再假定被调用函数又反过来调用了主调用函数。这第二个调用就被称为调用函数的递归，因为它发生在调用函数的当前执行过程运行完毕之前。而且，因为这个原先的主调用函数、现在的被调用函数在栈中处于较低的位置，有它独立的一组参数和自变量，原先的参数和变量将不受影响，所以递归能正常工作。

书归正传，回到阶乘的递归算法上来。假设要计算 5 的阶乘，通过前面设计的递归程序执行过程如下。

首先，在 main()函数中调用 fact()函数，将返回地址、参数、局部变量压入堆栈，如图 3-10 所示。

在 fact()函数中，判断 n 的值若不为 0，则递归调用 fact(4)，这时将函数的返回地址和参数压入堆栈，如图 3-11 所示。

将程序继续递归调用时，将 fact(3)、fact(2)、fact(1)逐步压入堆栈，如图 3-12 所示为将 fact(0)压入堆栈后栈的结构。



图 3-10

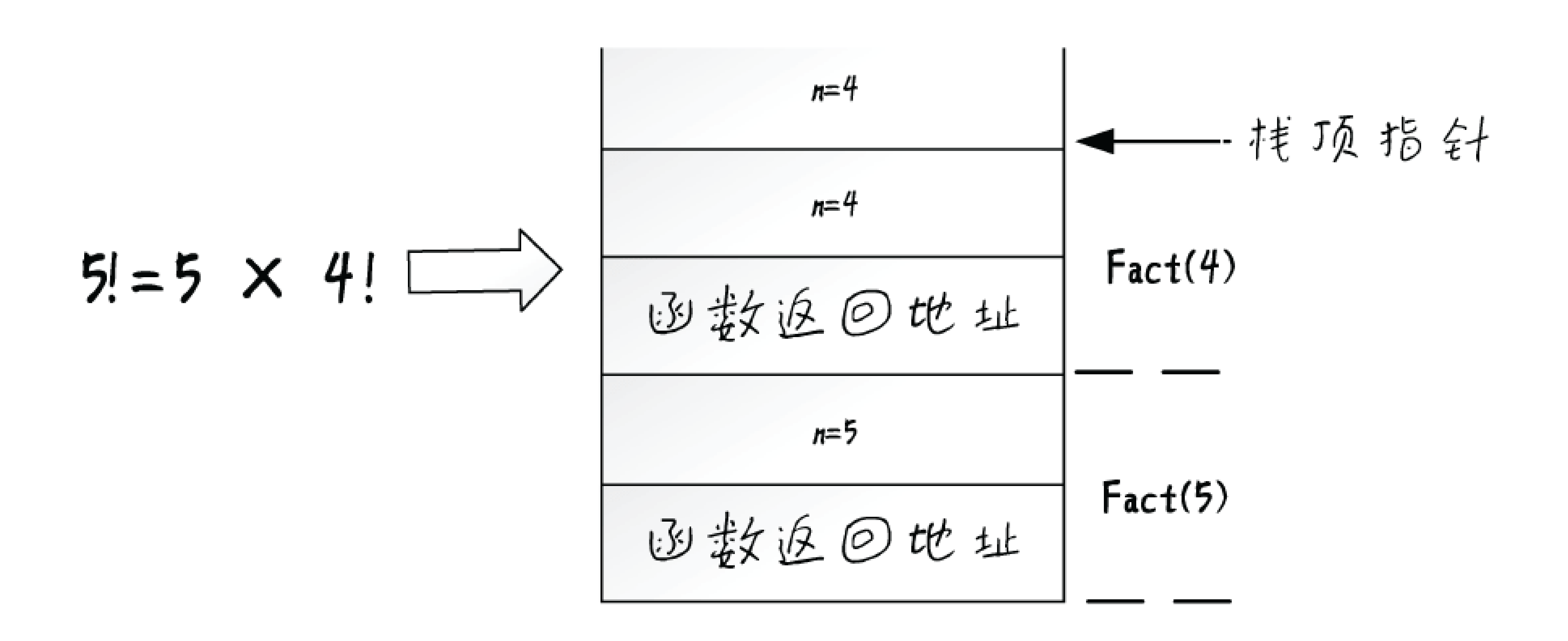


图 3-11

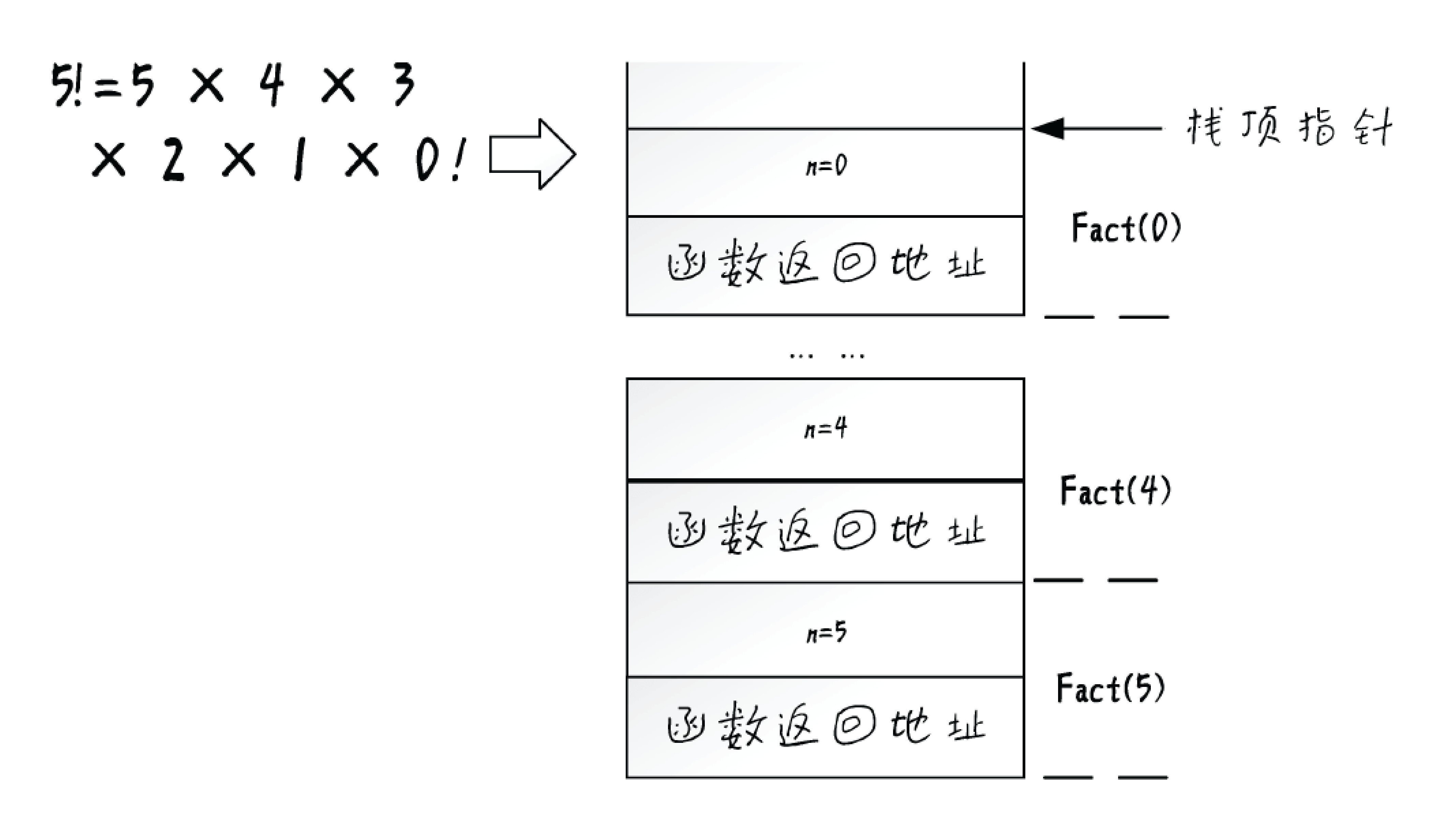


图 3-12

当调用 fact(0)时，达到阶乘递归算法的结束条件，这时结束 fact(0)函数调用，从堆栈中弹出该层的相关数据，并返回函数的结果 1。这时栈顶中保存的将是 fact(1)中的相关数据，如图 3-13 所示。

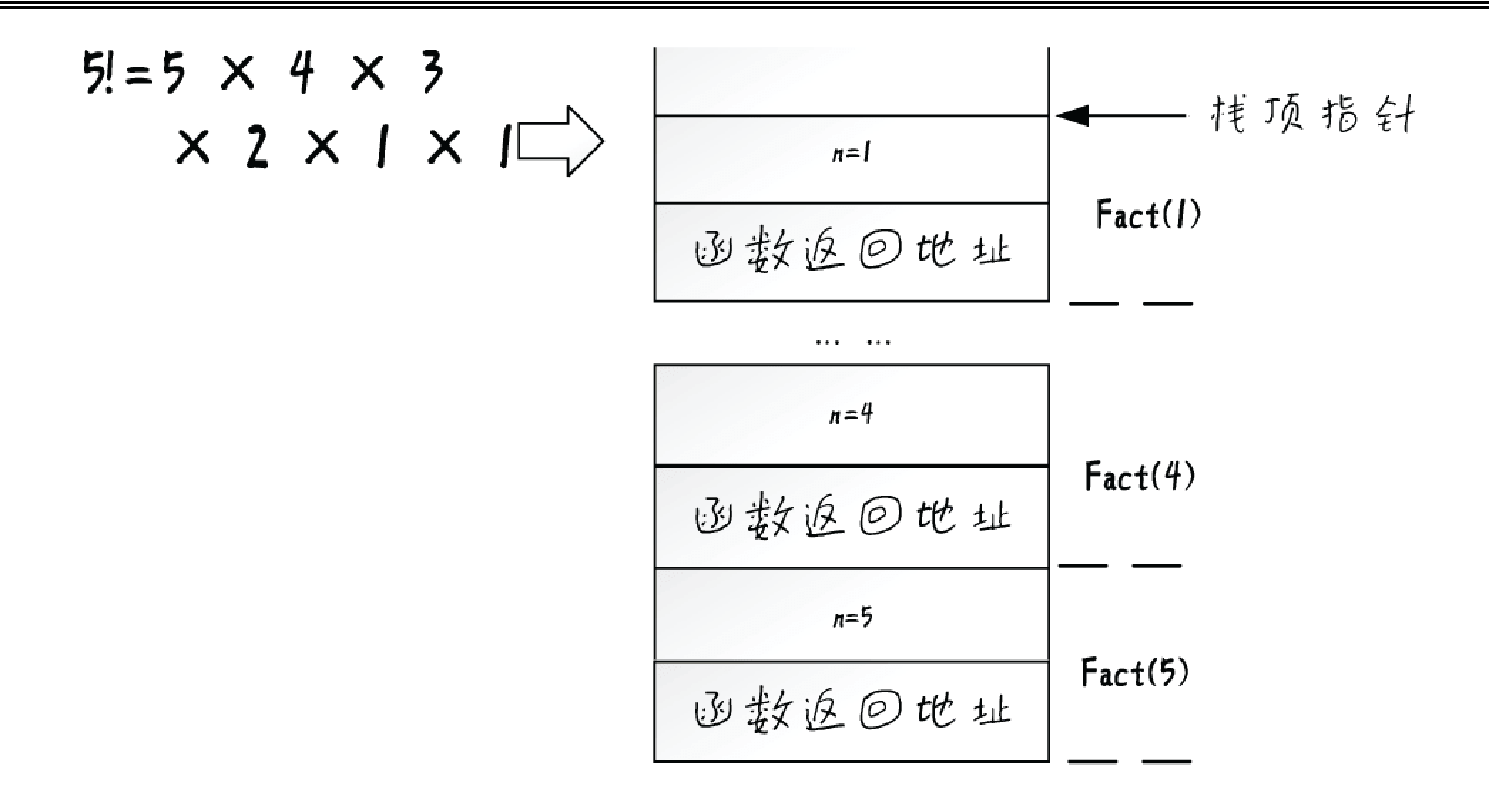


图 3-13

当递归函数逐层返回时，栈中压入的数据将逐步弹出，当弹出 fact(4)后的栈结果如图 3-14 所示。

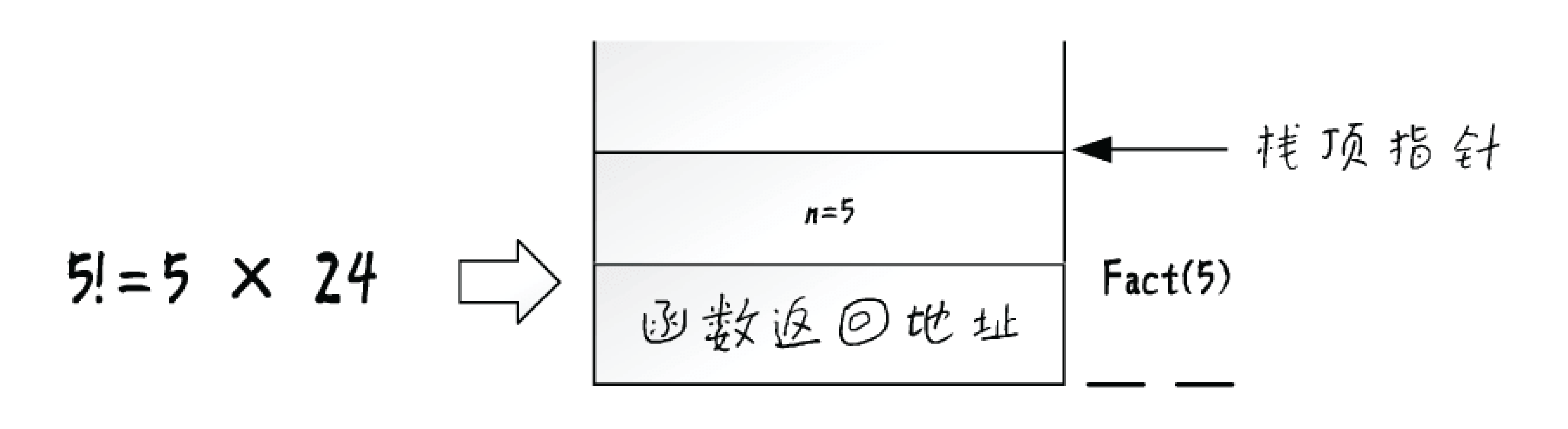


图 3-14

当函数 fact(5)返回时得到 5 的阶乘等于 120，同时从栈中弹出调用函数时的数据，完成整个递归调用。

3.2.4 递归的本质：缩小问题规模

递归式解决逻辑问题的基本思想是：把规模大的、较难解决的问题变成规模较小的、易解决的同一问题。规模较小的问题又变成规模更小的问题，并且小到一定程度可以直接得出它的解，从而得到原来问题的解。

用递归处理问题的过程，就是将问题规模逐步缩小的过程。

例如，在阶乘的递归运算中，就是将一个较大数的阶乘逐步缩小为一个较小数的阶乘，直到缩小到求 0 的阶乘为止。

在上节的例子中，用递归求解最大公约数的方法也同样如此，利用辗转相除法，在递归调用过程中逐步将被除数、除数缩小。

因此，在解决问题时，如果可以明确地将求解问题规则逐步缩小，就可考虑用递归

算法来实现。

利用递归算法编写的程序代码更简洁清晰，可读性更好。有的算法用递归表示比用循环表示简洁精练，而且某些问题，特别是与人工智能有关的问题，更适宜用递归方法，如八皇后问题、汉诺塔问题等。有的算法用递归能实现，而用循环却不一定能实现。

但是，也需要注意递归算法一个明显的缺点，每次递归调用时，需要将返回地址、参数等数据压入堆栈，也就是说，递归的内部实现要消耗额外的空间和时间。如果递归调用的层次太深，就可能导致堆栈溢出，从而使程序执行出错。

3.3　汉　诺　塔

汉诺塔问题是程序设计中的经典递归问题。

汉诺塔（又称河内塔）游戏是一个非常著名的益智游戏玩具，现在市面上卖的这个玩具外形如图 3-15 所示，这个游戏是从一个古老的传说演化而来。



图 3-15

3.3.1　古老的传说

相传在印度的贝纳雷斯，有座大寺庙，寺庙中有一块红木板，上面插着三根钻石棒，在盘古开天地，世界刚创造不久之时，神勃拉玛便在其中一根钻石棒上，放了 64 枚纯金的金片（圆盘），最大的圆盘在最底下，其余一个比一个小，依次叠上去。有一个叫婆罗门的门徒，不分日夜的向这座寺庙赶路，抵达后，就尽力将 64 枚纯金的圆盘移到另一根钻石棒上，在移动过程中，一次只能移动一个圆盘，且圆盘在放到钻石棒上时，大的不能放在小的上面。可利用中间的一根钻石棒作为辅助移动用。等到婆罗门完成这项工作，寺庙和婆罗门本身都将崩溃，世界将在一声霹雳中毁灭。

那么，世界毁灭会在哪一天呢？

经过计算，需要移动圆盘的次数是一个天文数字 18,446,744,073,709,551,615（64 个圆盘需要移动的次数为 2 的 64 次方）。假设 1 微秒进行一次移动，也需要约 60 万年的时间！当移动一个圆盘需花 1 秒钟时，完成所有圆盘的移动需要近 6000 亿年！何况移动一个圆盘的时间肯定不止 1 秒。

离世界毁灭还早得很，人们就根据这个传说演变成汉诺塔游戏，这个游戏的规则是：

- 将左侧柱子中从小到大放置的 8 个圆盘移到右侧的圆柱上。
- 每次只能移动一个圆盘。
- 小的圆盘只能放在大的圆盘之上。
- 可以借助另一个圆柱进行辅助移动。

那么，这个游戏该怎么玩呢？

假设有 8 个圆盘，如图 3-16 所示，将 A 柱中的最小圆盘移到 C 柱，再将 A 柱中第 2 个圆盘移到 B 柱。

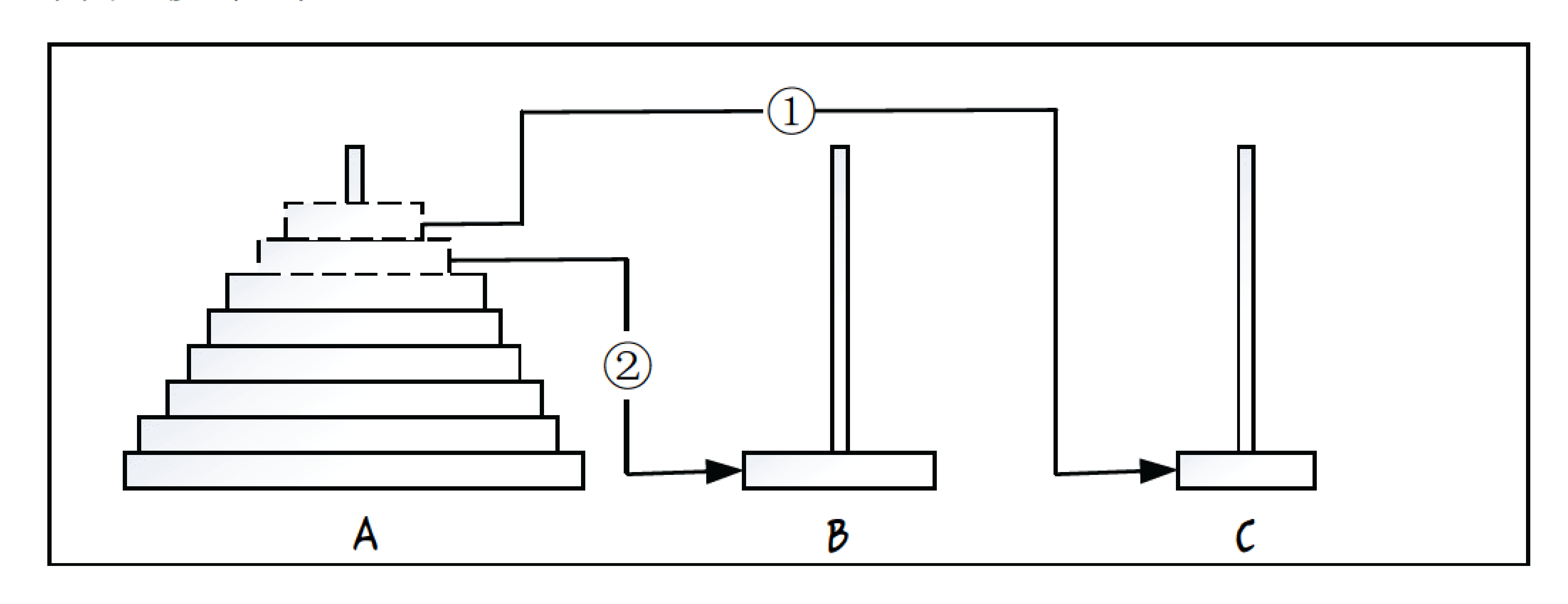


图 3-16

接下来该怎么办呢？又该怎么移动其余的圆盘呢？

3.3.2 从两个盘考虑

如图 3-16 所示的 8 个圆盘，移动起来就很麻烦，更别说 64 个圆盘的移动了。那么，我们先将问题简化一下，从两个圆盘开始考虑。即将 A 柱中的两个圆盘移到 C 柱，这个问题就简单了，如图 3-17 所示，共分 3 步即可完成任务：

（1）从 A 柱将小圆盘移到 B 柱。

（2）从 A 柱将下方大圆盘移到 C 柱。

（3）从 B 柱将小圆盘移到 C 柱，完成。

好，两个圆盘的移动通过 3 步就解决了，那么如果要从 A 柱移动 3 个圆盘到 C 柱又该怎么办呢？

由于已经知道将两个圆盘移动到另一个柱子时需要 3 个步骤，因此，这时就可以不

再考虑两个圆盘的移动了。

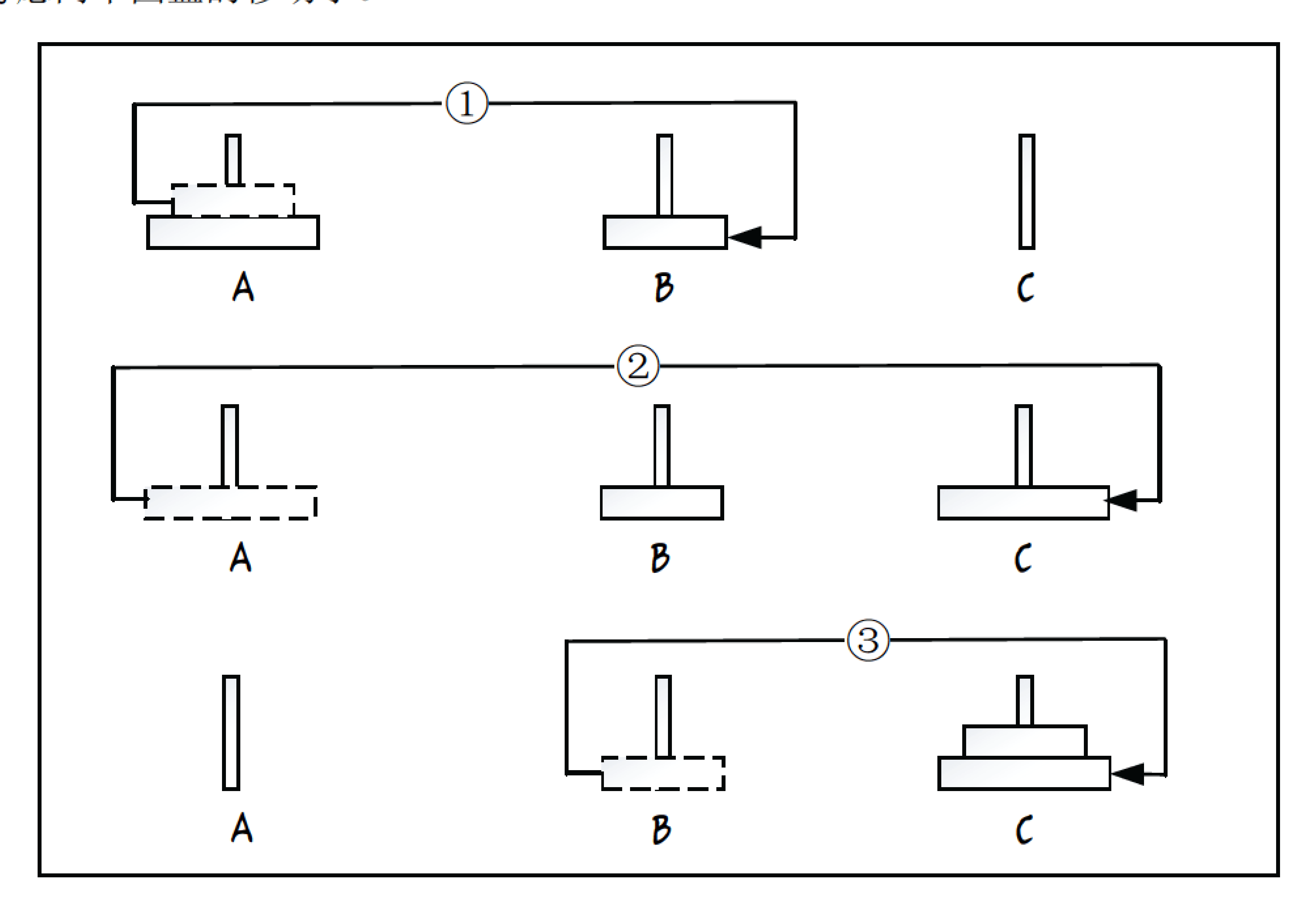


图 3-17

这时，可考虑将两个圆盘的移动看作一个整体，即 A 柱的 3 个圆盘中上面的两个圆盘看作为一个圆盘，下方最大圆盘作为一个圆盘，则 3 个圆盘的移动又可化解为两个圆盘的移动。这样，只需要 3 个步骤就可完成移动，如图 3-18 所示。

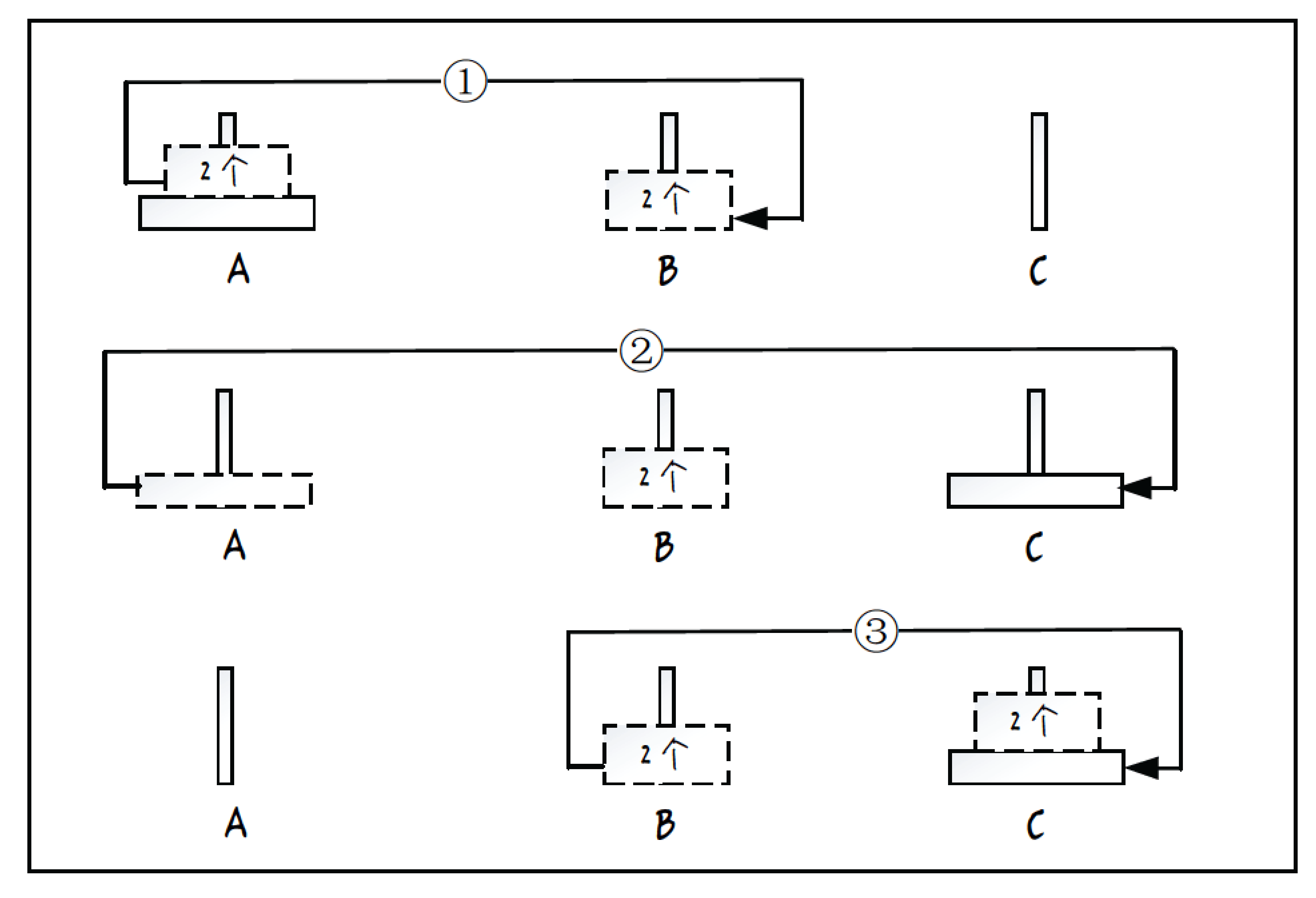


图 3-18

在图 3-18 中有以下 3 步：

（1）将 A 柱中的小圆盘（由上方两个圆盘组成）移到 B 柱。

（2）将 A 柱中的大圆盘（最下方的圆盘）移到 C 柱。

（3）将 B 柱中的小圆盘（由上方两个圆盘组成）移到 C 柱，完成。

在上面的第（1）步和第（3）步中，是将两个圆盘打包移动的。但是，按规则一次只允许移动一个圆盘，这里只是假想一次移动两个圆盘，在实际移动过程中，第（1）步和第（3）步中的每一步最终还是必须分解为 3 个步骤。

因此，如图 3-19 所示，在第（1）步中将上面的两个圆盘从 A 柱移到 B 柱时，需借助 C 柱作为辅助柱来进行移动。而在第（3）步中将 B 柱中的两个圆盘移到 C 柱时，需借助 A 柱作为辅助来进行移动。

将各动作分解后，可以看出，将 3 个圆盘从 A 柱移动到 C 柱时需要 7 步（第（1）步分解为 3 个步骤，第（3）步分解为 3 个步骤，再加上第（2）步中的 1 个步骤）。

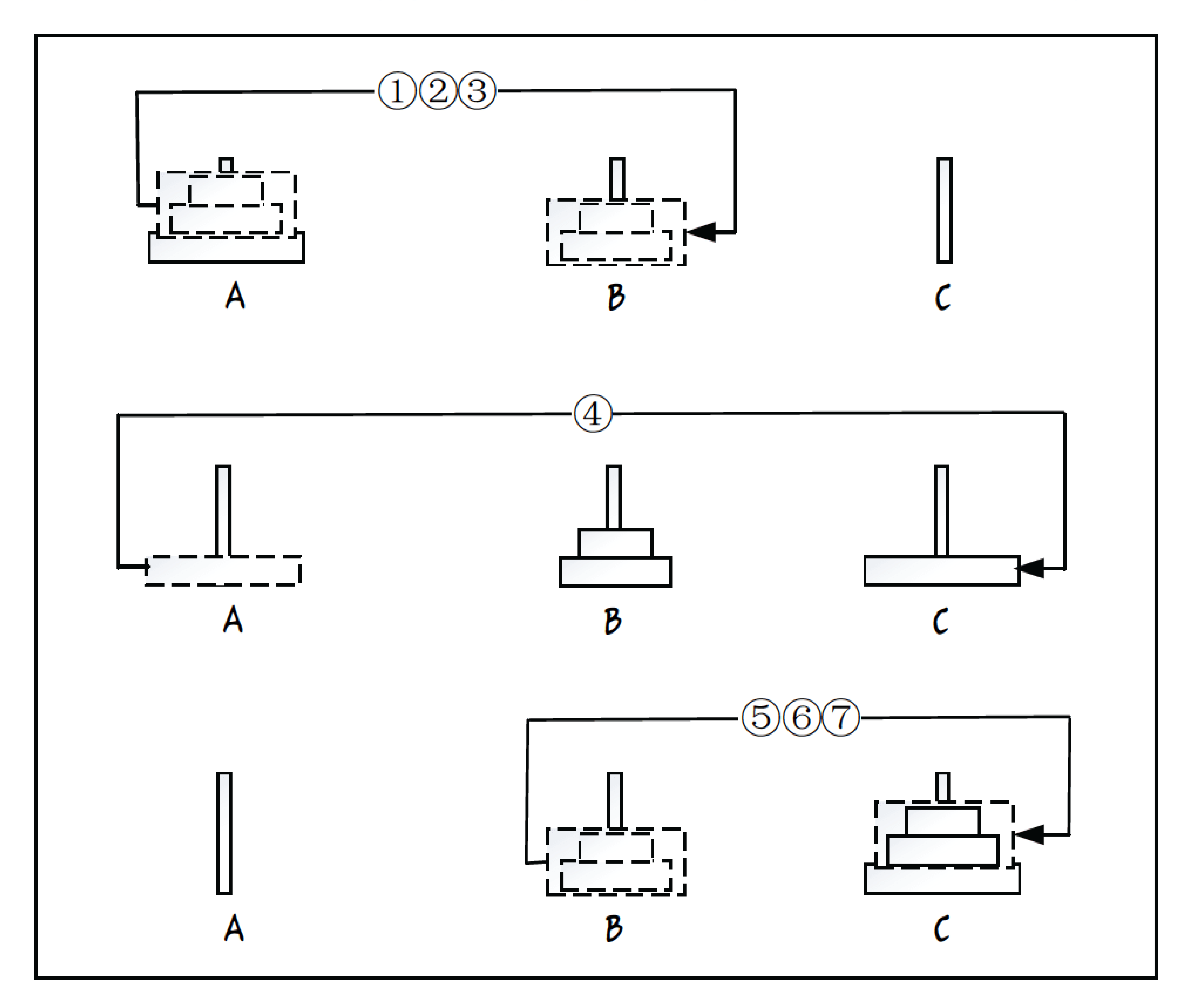


图 3-19

3.3.3 找出递归结构

经过上面的推算可看出，3 个圆盘的移动需要 7 步，那么移动 4 个圆盘呢？

与 3 个圆盘类似，对于 4 个圆盘也可分为 3 个大的步骤，如图 3-20 所示。

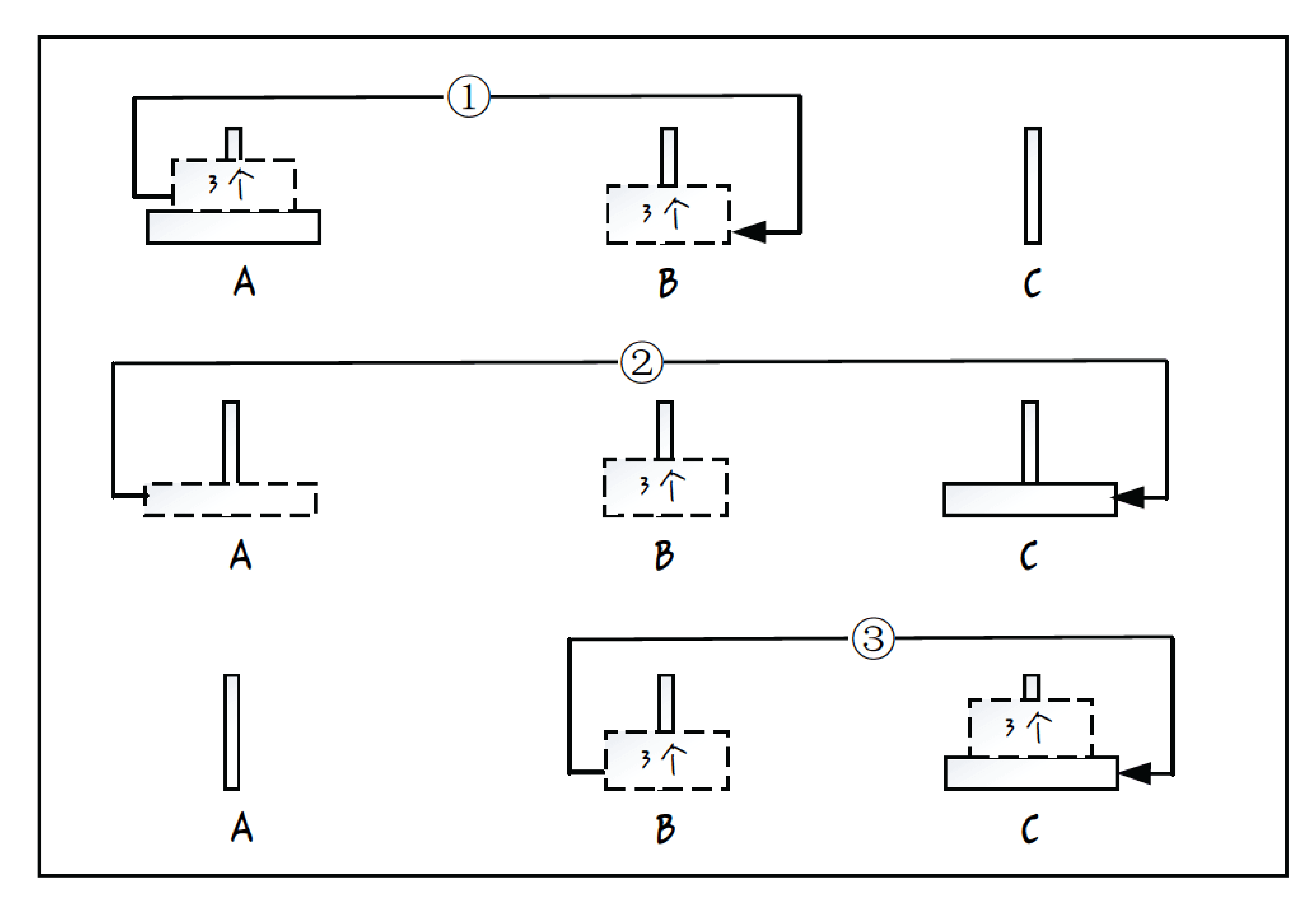


图 3-20

（1）将 A 柱中的小圆盘（由上方 3 个圆盘组成）移到 B 柱。

（2）将 A 柱中的大圆盘（最下方的圆盘）移到 C 柱。

（3）将 B 柱中的小圆盘（由上方 3 个圆盘组成）移到 C 柱，完成。

接着第（1）步、第（3）步需要移动 3 个圆盘，移动的步骤分别为 7 步，则将 4 个圆盘从 A 柱移动到 C 柱需要 7+1+7=15 步。

对比图 3-18 和图 3-20 可以看出，如果不考虑第（1）步和第（3）步移动圆盘的数量，这两个图完全一样。也就是说，不管移动多少个圆盘，其实，移动的操作相似。

既然解决问题时具有相似步骤，并且可逐步缩减问题规模，就可考虑使用递归算法来求解。通过上面对移动 3 个圆盘和 4 个圆盘时的分析，可总结出汉诺塔的递归结构，对于移动 n 个圆盘的汉诺塔，可分解为 3 步，如图 3-21 所示。

（1）移动（n-1）个圆盘。

（2）移动第 n 个圆盘。

（3）移动（n-1）个圆盘。

而“移动（n-1）个圆盘”又可继续分解，直至分解到只剩一个圆盘时，直接移动到目标柱为止。

根据图 3-21 所示的递归结构，可将汉诺塔问题的递归求解法分解为以下步骤。

（1）如果只有一个圆盘，则把该圆盘从 A 棒移动到 C 棒，完成任务。

（2）如果圆盘数量 n>1，移动圆盘的过程可分为 3 步。

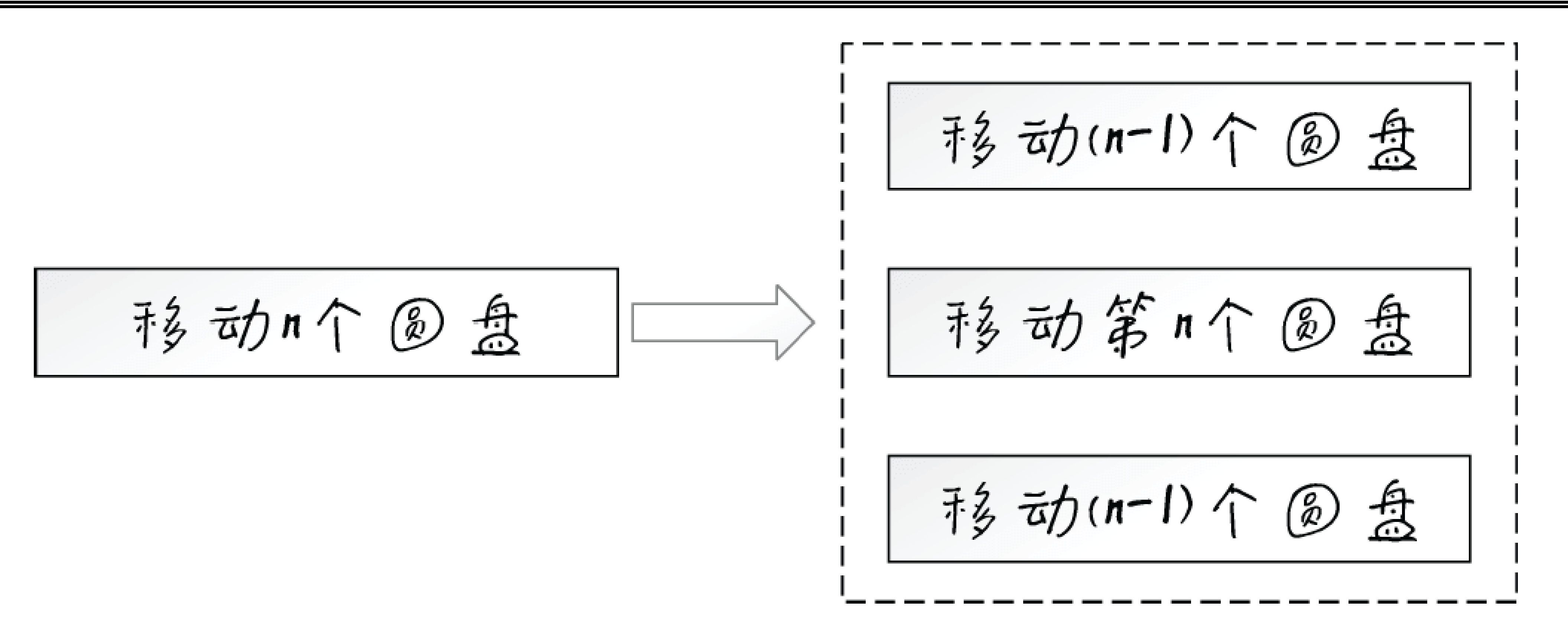


图 3-21

（1）将 A 棒上的 n-1 个圆盘移到 B 棒上。

（2）将 A 棒上的一个圆盘移到 C 棒上。

（3）将 B 棒上的 n-1 个圆盘移到 C 棒上。

3.3.4 实现程序

将递归结构找出来以后，编写递归程序就很简单了，下面的代码就是用 C 语言编写的实现汉诺塔的递归程序。

```
#include <stdio.h>
long count;                                    //全局变量，记录移动的次数

void hanoi(int n,char a,char b,char c)         //a 移到 b，用 c 作为临时柱
{
    if(n==1)
    {
       printf("第%d 次，%c 柱-->%c 柱\n",++count,a,c);
    }
    else
    {
        hanoi(n-1,a,c,b);                      //递归调用
        printf("第%d 次，%c 柱-->%c 柱\n",++count,a,c);
        hanoi(n-1,b,a,c);                      //递归调用
    }
}

int main()
{
    int h;                                     //圆盘数量

    printf("请输入 A 柱汉诺塔圆盘的数量:");
    scanf("%d",&h);
```

```
    count=0;
    hanoi(h,'A','B','C');

    getch();
    return 0;
}
```

在以上程序中，函数 hanoi()是一个递归调用的函数。这个函数共有 4 个参数，第 1 个参数表示要移动的圆盘数量，第 2～4 个参数表示移动的源位置、临时位置、目标位置。例如，以下调用 hanoi()函数的形式：

```
    hanoi(h,'A','B','C');
```

表示有 *h* 个圆盘要从 A 柱移到 C 柱，B 柱作为临时辅助用的柱。

在递归调用时，需要搞清楚每次移动圆盘的源位置和目标位置。例如，若要将 *h* 个圆盘从 A 柱移到 C 柱，则需要将（*h*-1）个圆盘先从 A 柱移到 B 柱（借助 C 柱作为临时辅助），完成这个操作需按以下方式调用 hanoi()函数：

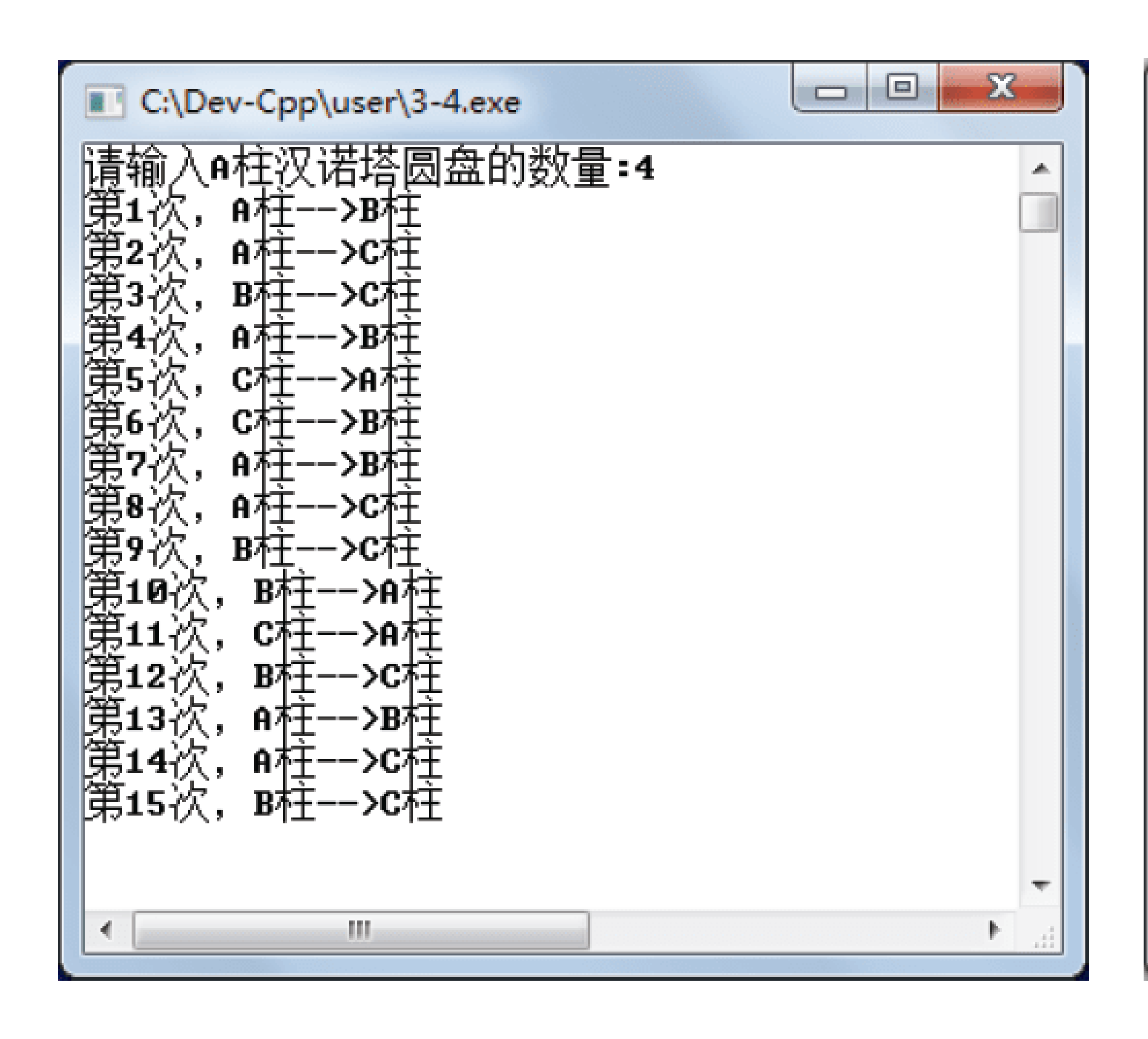
```
hanoi(h-1, 'A','C','B');
```

接着将第 *h* 个圆盘从 A 柱移到 C 柱。

然后将临时放在 B 柱的（*h*-1）个圆盘移到 C 柱（借助 A 柱作为临时辅助），这时需按以下方式调用 hanoi()函数：

```
hanoi(n-1, 'B','A',C');
```

执行以上程序，当输入 4 个圆盘时，移动圆盘的过程如图 3-22 左图所示，共需 15 次可完成任务。当输入 6 个圆盘时，移动圆盘的过程如图 3-22 右图所示，共需要 63 次可完成任务。



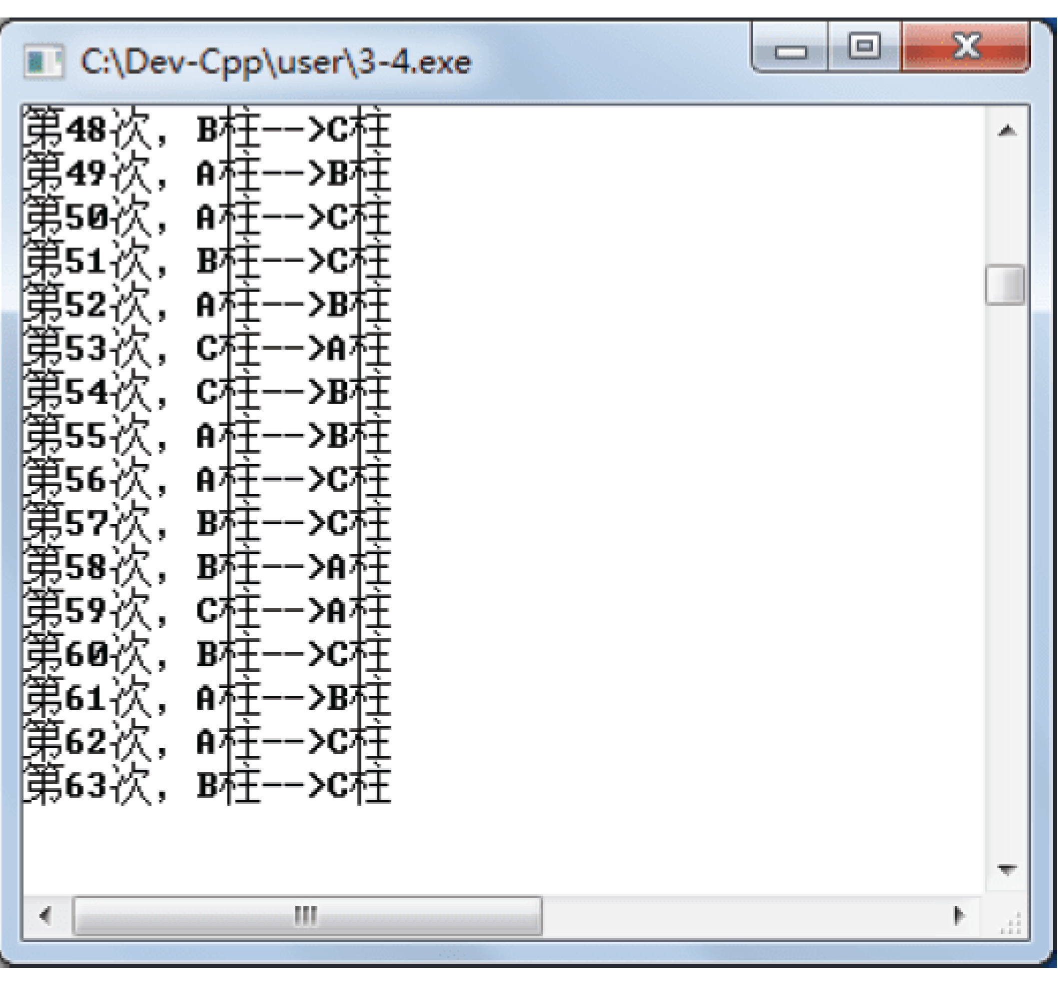


图 3-22

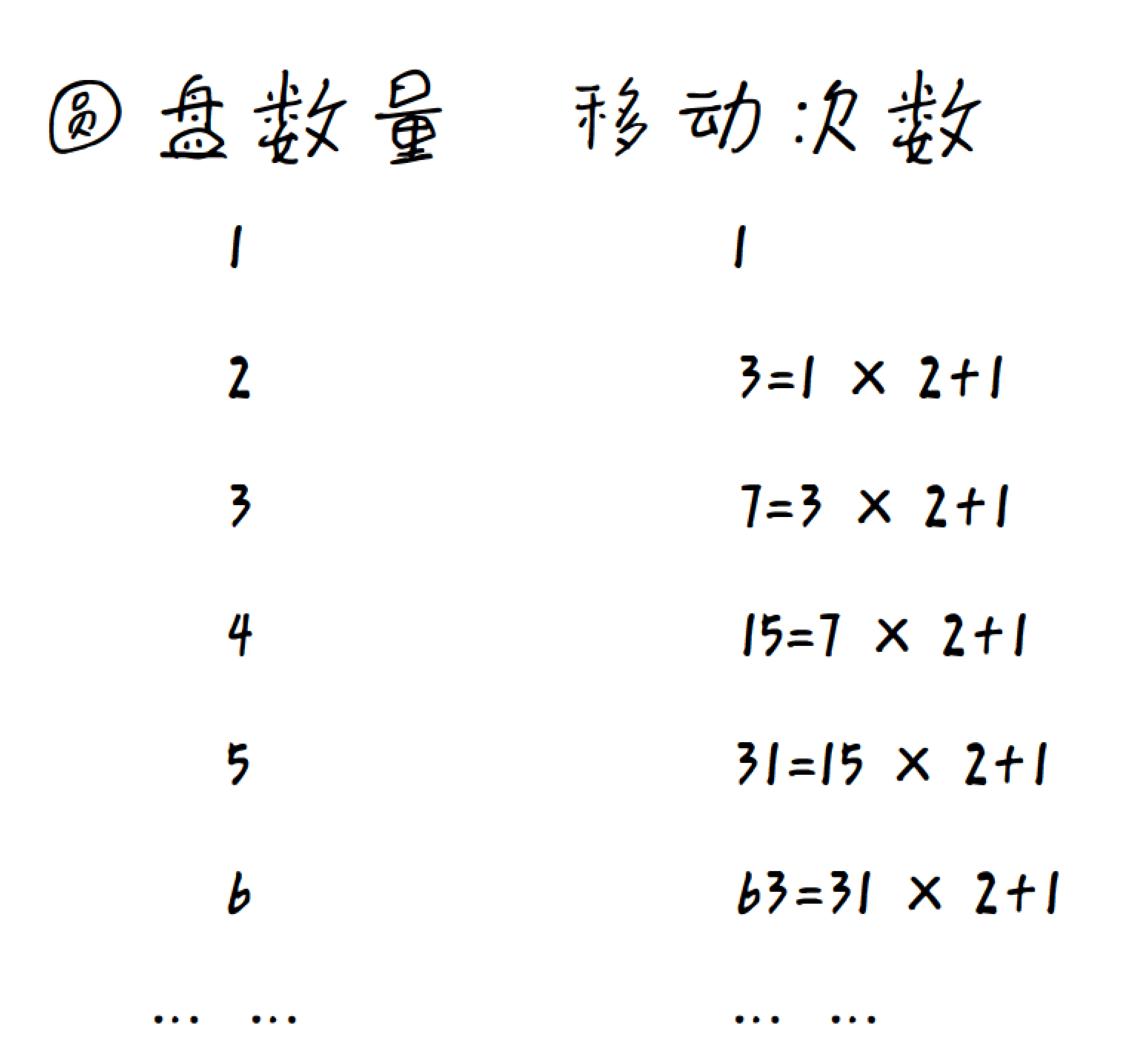
3.3.5　究竟需要移动多少次

根据我们前面手工推算移动次数，以及图 3-22 所示计算机程序运行得出的移动次数，可发现当需要移动的圆盘数量为 1、2、3、4、5、6……时，移动圆盘的次数分别为：

圆盘数量：1、2、3、4、5、6 …

移动次数：1、3、7、15、31、63 …

可以看出，圆盘数量按自然数序列递增，但移动次数却呈接近倍数的方式递增：



按上面列表中计算的移动次数，每次都是在上一次移动次数的基础上乘以 2 再加 1，如果要直接计算移动 n 个圆盘需要多少次移动，就需要逐个推算。其实，仔细分析上面的列表，可发现如下规律：

$$\text{移动次数} = 2^n - 1$$

这样，当要移动 64 个圆盘，就需要：

$$2^{64} - 1 = 18\ 446\ 744\ 073\ 709\ 551\ 615$$

3.4　斐波那契数列

“斐波那契数列（Fibonacci）”是由意大利数学家列昂纳多·斐波那契发明的，因此取名为斐波那契数列。在自然界中，很多现象都符合斐波那契数列，例如，植物的叶、鳞片、花、茎等排列中，都可发现这种规律。那么，斐波那契数列有什么规律？让我们先从兔子的繁殖说起。

3.4.1　兔子的家族

先来看著名斐波那契数列的例子：兔子家族。

意大利数学家列昂纳多·斐波那契在所写的《算盘书》中提出了下面的问题：

有小兔一对，如果它们第 2 个月成年，第 3 个月生下一对小兔，以后每月生产小兔一对，而所生的小兔亦在第 2 个月成年，第 3 个月生产另一对小兔，此后也每个月生一对小兔。问一年后共有多少对兔子（假设每产一对兔子必须为一雌一雄，而所有兔子都可以相互交配，并且没有死亡）。

要计算出一年后共有多少对兔子，没有公式可直接计算出来。根据题意，每月兔子的数量与上月兔子和上上月兔子的数量有很大关系，可通过前两个月的兔子数量推算出当月兔子的数量。

3.4.2　从最初几月数据中找规律

从第 1 个月开始向后推算：

第 1 个月，只有一对小兔子。

第 2 个月，小兔子成年，仍然只有一对兔子。

第 3 个月，有一对成年的兔子，成年的兔子生产一对小兔子，共有两对兔子。

第 4 个月，有两对成年的兔子，以及第 3 月 1 对成年兔子所生产的一对小兔子，共有三对兔子。

第 5 个月，有三对成年兔子，以及第 4 个月的两对成年兔子所生产的两对小兔子，共有 5 对兔子。

前面 5 个月兔子繁殖的过程如图 3-23 所示。

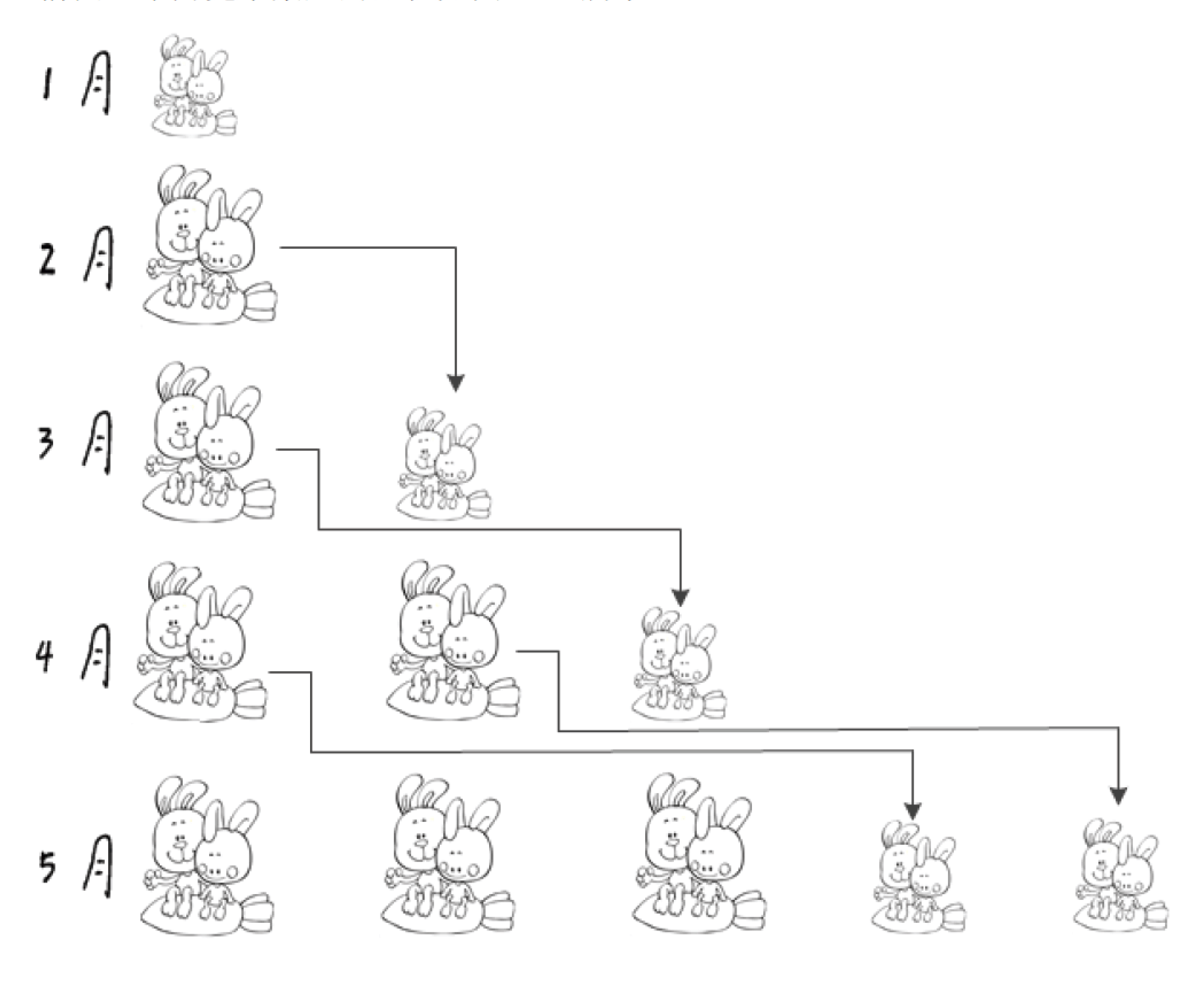


图 3-23

设 F(*n*)表示第 *n* 个月兔子的总数，则图 3-23 所示各月兔子数量可用以下算式来表示：

1 月： $F(1)=1$

2 月： $F(2)=1$

3 月： $F(3)=F(2)+F(1)=1+1=2$

4 月： $F(4)=F(3)+F(2)=2+1=3$

5 月： $F(5)=F(4)+F(3)=3+2=5$

按上面列出的算式，可以很快推导出 1 年（共 12 个月）后兔子的总数，具体算式如下：

6 月 ： $F(6)=F(5)+F(4)=5+3=8$

7 月 ： $F(7)=F(6)+F(5)=8+5=13$

8 月 ： $F(8)=F(7)+F(6)=13+8=21$

9 月 ： $F(9)=F(8)+F(7)=21+13=34$

10 月 ： $F(10)=F(9)+F(8)=34+21=55$

11 月 ： $F(11)=F(10)+F(9)=55+34=89$

12 月 ： $F(12)=F(11)+F(10)=89+55=144$

也就是说，1 年后兔子的总数为 144 对。

也可将各月的成年兔子与小兔子的数量制作成一个表格，如下所示。

月份	1	2	3	4	5	6	7	8	9	10	11	12
小兔	1	0	1	1	2	3	5	8	13	21	34	55
成年兔	0	1	1	2	3	5	8	13	21	34	55	89
总对数	1	1	2	3	5	8	13	21	34	55	89	144

各月兔子的总数量组成一个列表，如下所示。

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

3.4.3　斐波那契数列

在上面列出的兔子繁殖过程中，每月兔子的数量组成一个序列，这个序列就称为斐波那契数列。这个数列从第 3 项开始，每一项都等于前两项之和。在数学上，斐波那契数列可以用递归的方法来定义：

$$F_0=0$$

$$F_1=1$$

$$F_n=F_{n-1}+F_{n-2}$$

在上面的算式中 F_0 不是第一项，而是第 0 项，只是递归而定义的一个结束项。

在前面计算兔子总数时，我们是从第 1 个月开始逐月向后推导，如果要计算的项 n 的数据很多，就需要很多个步骤进行推导。好在我们找到了斐波那契数列的递归定义方式，在上面的 3 个算式中，F_n 的值由 F_{n-1} 和 F_{n-2} 得出，而 F_{n-1} 和 F_{n-2} 又可由它们的前两项得出。也就是说，每次递归调用，都将问题规模缩小，当递归到第 1 项和第 0 项时，返回确定的值，这就可使递归调用结束。

只是，在这里的递归调用与本章前面例子中不一样，当计算第 n 项数据时，需要递归调用（n-1）项和（n-2）项。

根据以上分析，可用 C 语言编写如下程序计算出斐波那契数列中的前 n 项数据：

```
#include <stdio.h>

int fibo(int n)
{
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
        return fibo(n-1)+fibo(n-2);
}

int main()
{
    int f,i;
    printf("请输入斐波那契数列的数量：");
    scanf("%d",&f);

    printf("斐波那契数列：\n");
    printf("F(0)=0\n");
    printf("F(1)=1\n");
    for(i=2;i<=f;i++)
    {
        printf("F(%d)=F(%d)+F(%d)=%d+%d=%d\n",
          i,i-1,i-2,fibo(i-1),fibo(i-2),fibo(i));
    }

    getch();
    return 0;
}
```

执行以上程序，输入 12，表示生成 12 项斐波那契数列，得到如图 3-24 所示的结果。

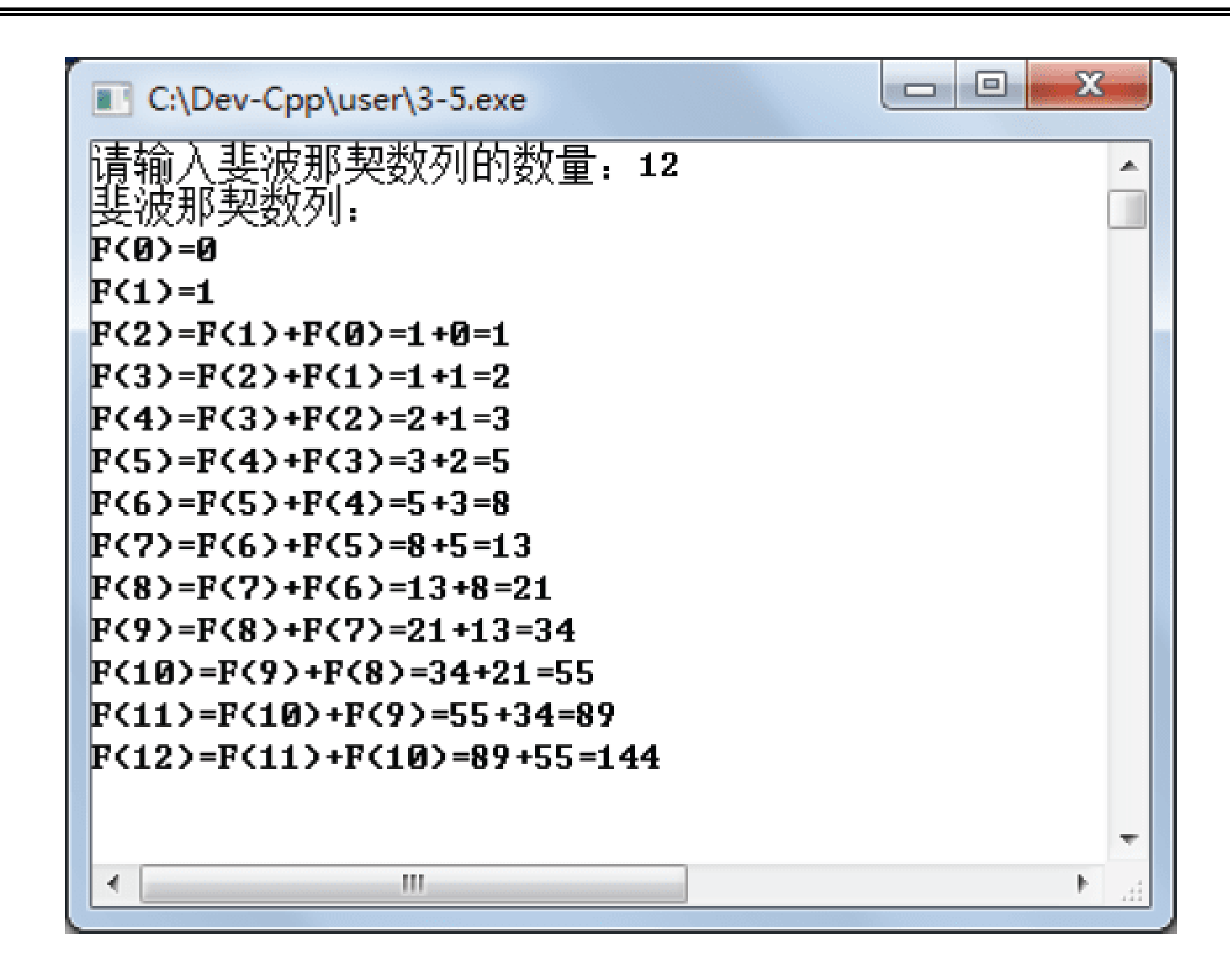


图 3-24

使用上面的程序，用递归方式推导出第 n 项的斐波那契数，如果 n 的值很大，这个推导过程就比较繁琐。其实，用初等代数解法也可推导出计算第 n 项斐波那契数的公式，具体推导过程这里就不列出来了，具体计算公式如下：

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

通过上面的公式就可以计算第 n 项斐波那契数。

如果觉得以上公式计算起来还是比较麻烦的话，可以使用以下公式得到一个大约数值：

$$\begin{aligned} F_n &\approx \frac{1}{\sqrt{5}}\cdot\left[\frac{1}{2}\left(1+\sqrt{5}\right)\right]^n \\ &\approx 0.4472135955 \times 1.618033988745^n \end{aligned}$$

例如，若取 n=12，则：

$$\begin{aligned} F_{12} &\approx 0.4472135955 \times 1.6180339887\ 45^{12} \\ &\approx 0.4472135955 \times 321.996894\ 368296 \\ &\approx 144.001388\ 87 \end{aligned}$$

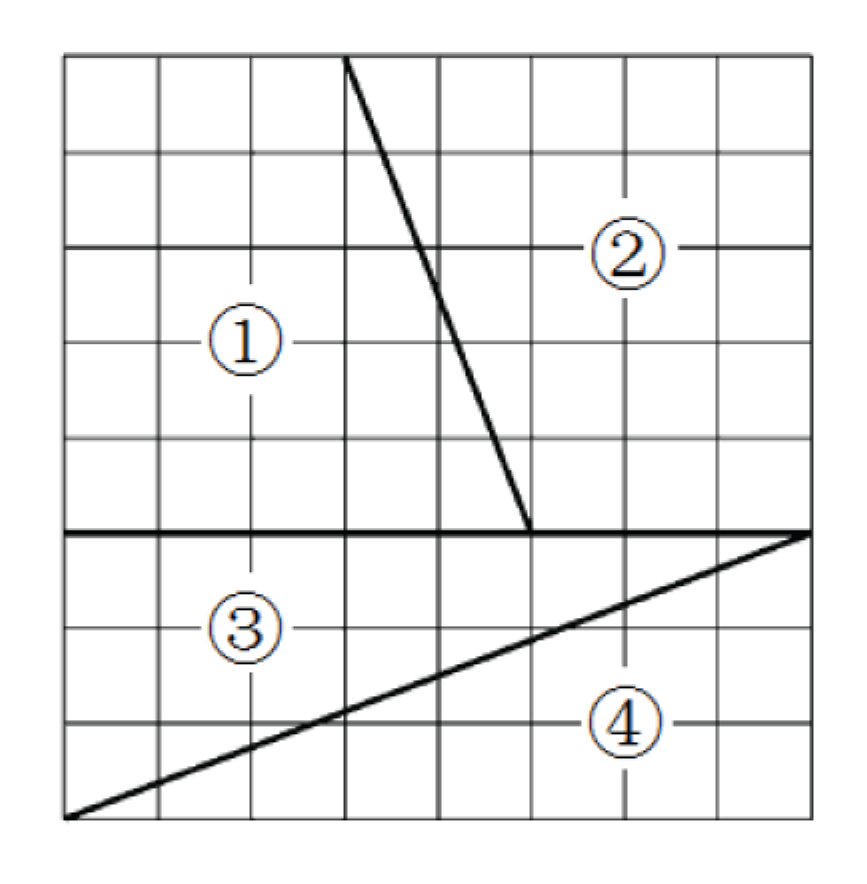
3.4.4　神奇的魔八方

最后，来看一个斐波那契数在魔术中的应用：“魔八方”。

一位魔术师拿着一块边长为 8 分米的正方形地毯，对观众朋友说："我能把这块地毯分成 4 小块，再重新缝成一块长 13 分米，宽 5 分米的长方形地毯，你相信吗？"

观众中有人开始计算了：边长 8 分米的正方形，其面积为 64 平方分米；而边长 13 分米，宽 5 分米的长方形，其面积为 65 平方分米。两者面积相关 1 平方分米，肯定不行啊。

魔术师之所以称为魔术师，肯定有他的绝活，结果他还真的拼出了长 13 分米、宽 5 分米的地毯。如图 3-25 所示，就是魔术师的裁剪及拼接过程，左图将正方形裁剪成了 4 部分，右图是由这 4 部分拼接而成。



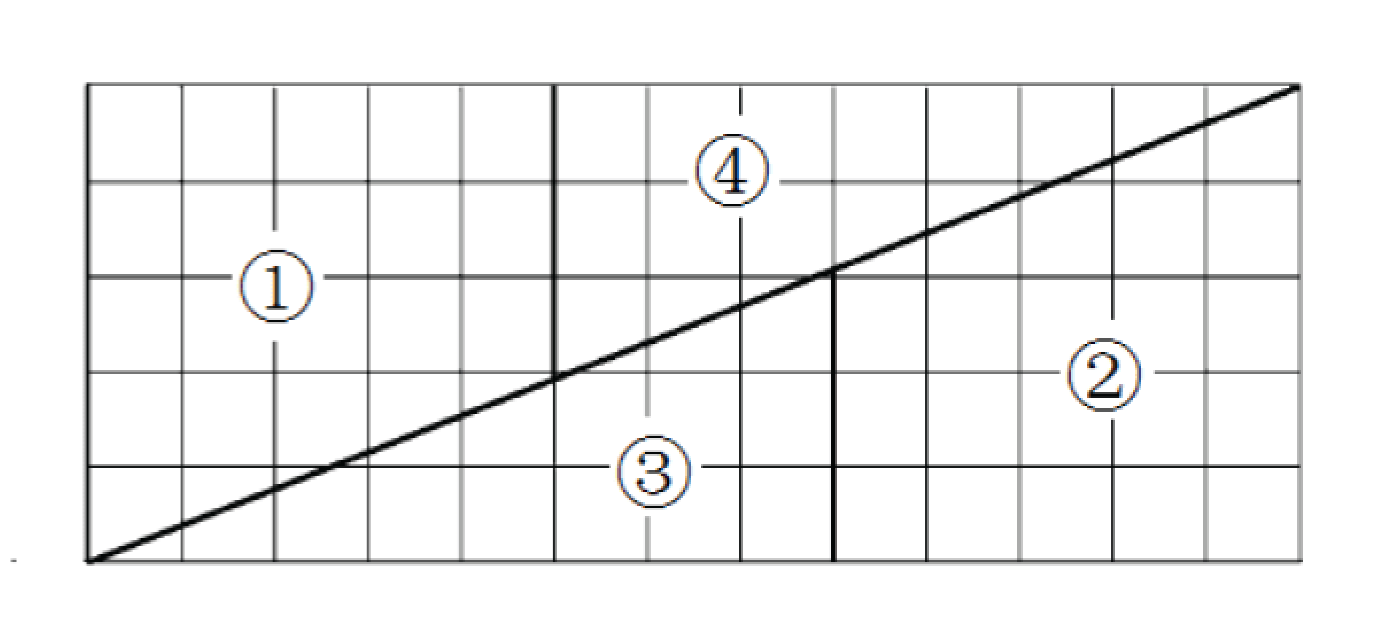


图 3-25

为什么经过一裁一拼，地毯的面积就增加了 1 平方分米呢？为了分析这个问题，将图 3-25 右图所示的长方形以左下角为原点制作一个坐标系，如图 3-26 所示。

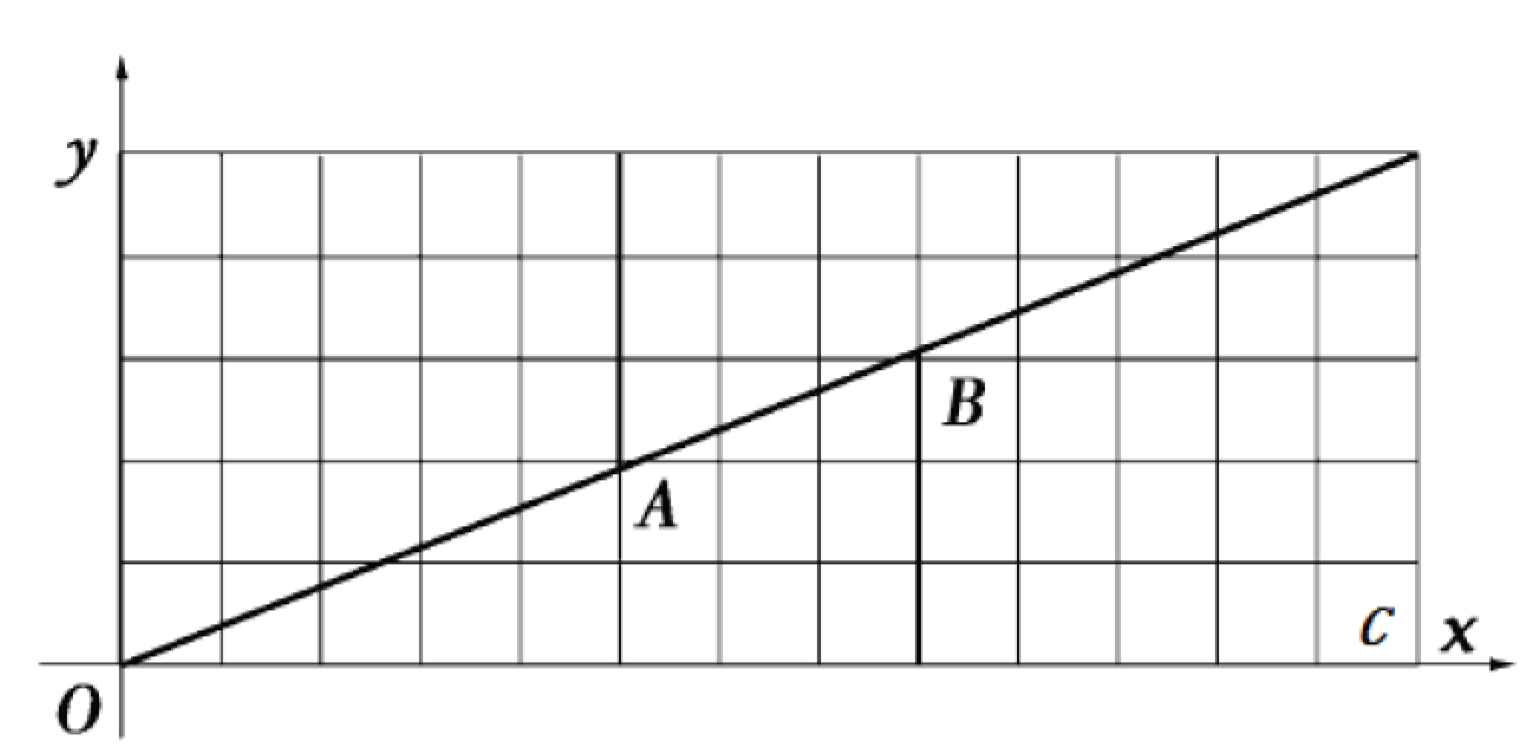


图 3-26

在图 3-26 所示坐标系中，O 点为原点，则 A 点的坐标为（50，20），B 点的坐标为（80，30），则∠AOC 的正切为 2/5，而∠BOC 的正切为 3/8，显然 OA 与 OB 不在一条直线上。也就是说，实际上后来缝成的地毯有条细缝，面积刚好就是 1 平方分米。

那么，是不是所有正方形都可以进行这种方式的变化呢？

先来计算一下，设边长为 *n* 的正方形是否可以进行这种魔八方的变化。

根据图 3-25 所示的裁剪和拼接方式，可知道拼接后的矩形的长度为：

$$n+y \qquad (0<y<n)$$

由于拼接后的长方形面积比正方形的面积多 1，则可得到以下公式：

$$y(n+y)=n^2+1$$

设：

$$n=x+y$$

代入上面的方程，可得到如下方程：

$$y(n+y)=n^2+1$$

$$\Downarrow$$

$$y(x+y+y)=(x+y)^2+1$$

$$\Downarrow$$

$$x^2+xy-y^2+1=0$$

对于这个二元二次方程，由于只有一个方程式，属于不定方程，将这个不定方程中的 x、y 的值求出来，即可得到正方形的边长。对于不定方程，我们可以先计算出 y 的值在 100 以内时的各种整数解，如下表所示。

x	1	3	8	21	55
y	2	5	13	34	89
n	3	8	21	55	144

可以看出，x、y 的值构成了斐波那契数列：1、2、3、5、8、13、21、34、55、89、……，也就是说，正方式的边长 n 应为斐波那契数列的前两项之和。例如，边长为 8、13、21 这些正方形都可以按魔八方的形式对图形进行裁剪和拼接，以使其面积增大。

第 4 章　排列组合——让数选边站队

所谓排列，就是指从给定个数的元素中取出指定个数的元素进行排序。组合则是指从给定个数的元素中仅仅取出指定个数的元素，不考虑排序。排列组合的中心问题是研究给定要求的排列和组合可能出现的情况总数。

4.1　把所有情况都列出来

数学是我们日常生活中不可或离的知识，例如，购买商品的价格、数量，接待来访客户的数量，与客户签订合同的数量和金额等，这些都用了到最基本的数学知识计数。

计数，是我们从幼儿园、小学就开始学习的最基本数学知识，可是，到现在为止，你会计数吗？能正确的计数吗？

4.1.1　从 0 还是 1 开始

计数还会出错？好吧，那我们来看一个例子。

小张每天从家里到办公室去上班乘坐地铁都要购买 3 元的票，再多坐一站都需要购买 4 元的票（根据当地地铁公司的票价规定，乘坐 10 个区间的票价为 3 元）。请问，小张早上去上班要路过多少个车站？

你可能觉得这太简单了吧，这也是一个问题？

根据题意，小张购买的是 3 元的车票，再多坐一站都要购买 4 元的车票，说明小张坐的车站是 3 元内的最多车站数，即 10 站。

所以，小张早上上班路过的车站是 10 个站。

可是，这个答案却是错误的！

正确的答案应该是要经过 11 个车站。为什么是这样呢？看图 4-1 就可得到答案。

在图 4-1 中，经过的区间用圆圈中的数字表示，经过的车站的编号用下方的数字表示。从图中可看到，小张从地铁站 Home 出发，经过 10 个区间到达 Office 站，包括 Home 站在内，一共经过了 11 个地铁站，其中 Home 的编号为 0。

这是从 0 开始计数的表示方式，与我们平常从 1 开始计数有些不同，所以很多人都容易在这里犯错。

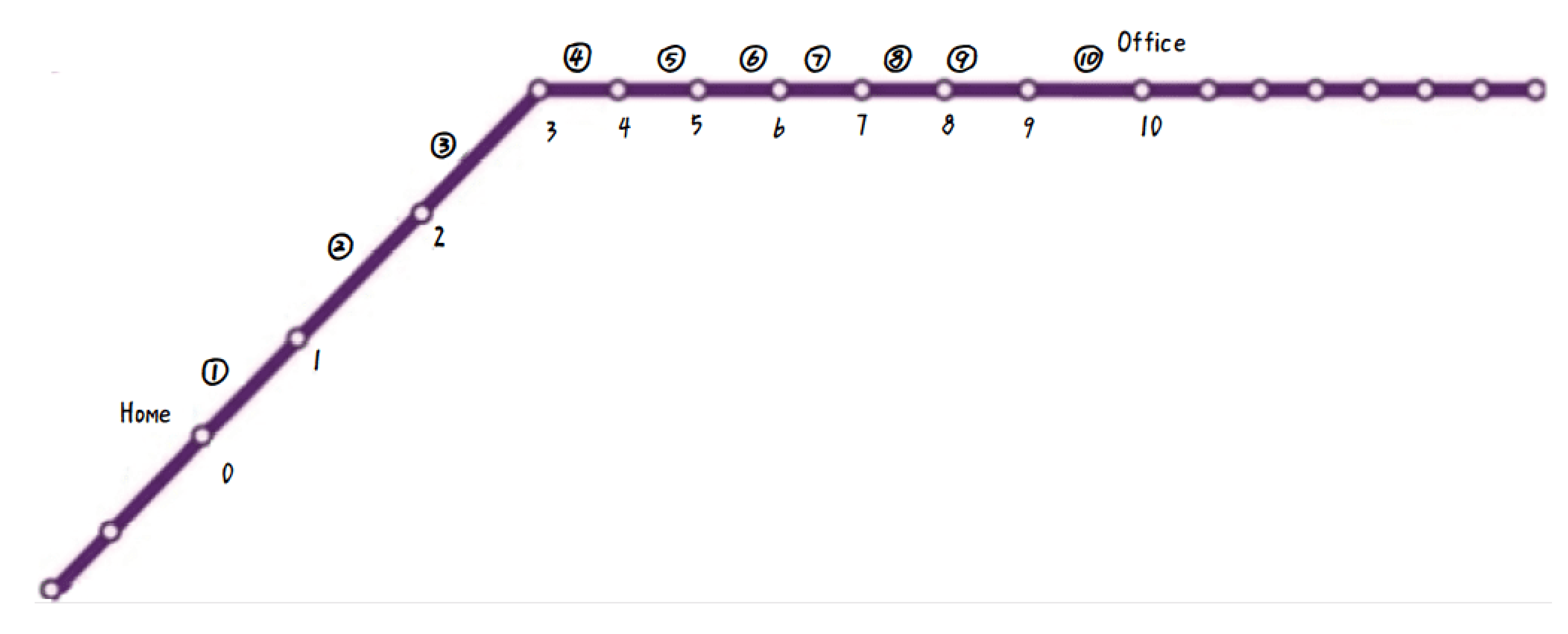


图 4-1

对于从 0 开始计数的情况，最后的数据数量应是计数值加上 1，即：

序号为n的数据D(n)是第n+1个数据

其实，从 0 开始计数这种方式程序员应该是比较熟悉的，在 C、Java 等流行的程序设计语言中，数组的下标都是从 0 开始计数，例如，有以下 C 语言程序：

```
#include <stdio.h>

int main()
{
    int i, arr[10]={1,2,3,4,5,6,7,8,9,10};

    for(i=0;i<10;i++)
    {
        printf("arr[%d]=%d\n",i,arr[i]);
    }

    getch();
    return 0;
}
```

执行以上 C 语言程序，可得到如图 4-2 所示的结果。

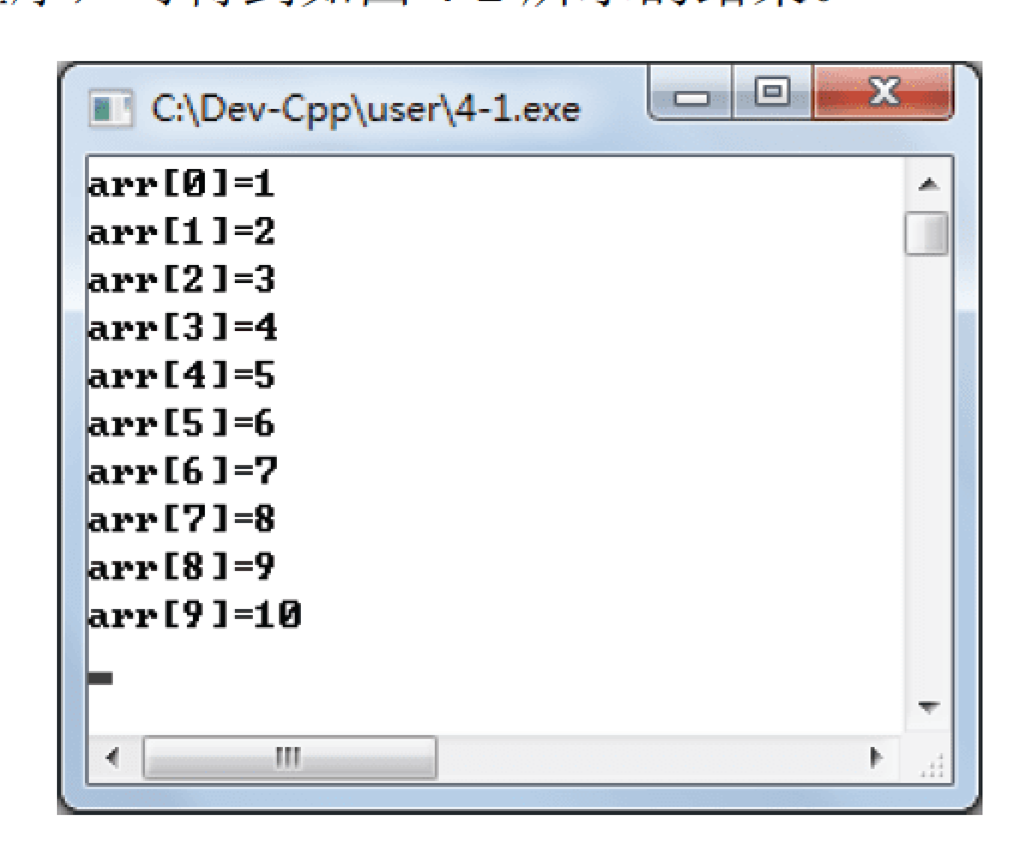


图 4-2

从图 4-2 所示的执行结果可看到，数组元素的序号从 0 开始，程序中定义的数组具有 10 个元素，其序号为 0～9。

因此，如果是以 0 开始计数，要注意：

以0开始计数时，

不要遗漏第0号数据。

4.1.2 赛程安排

赛程安排也是一个计数问题。

在比赛的赛程安排中，不能漏掉参赛的每一位选手，也不能为某一位选手安排重复的比赛。这就是计数中需要解决的两个问题：遗漏和重复。

某学校举行乒乓球比赛，在初赛阶段设置为单循环赛，设有 n 位选手参赛，每位选手要与其余每位选手进行一场比赛，然后按积分排名选拔进入决赛的选手。这个比赛的赛程应该怎么安排呢？

为了简化，先以 5 位选手参加比赛为例来进行分析。

首先，每位选手都要与其余的每位选手进行一场比赛，制作一张二维表，将参赛选手分别排在行和列中。在纵横方向上都可以设置与之进行比赛的选手，得到如图 4-3 所示比赛对阵表。

	甲	乙	丙	丁	戊
甲	×	√	√	√	√
乙	√	×	√	√	√
丙	√	√	×	√	√
丁	√	√	√	×	√
戊	√	√	√	√	×

图 4-3

在图 4-3 所示的比赛对阵表中，选手自己不能与自己比赛（如甲与不能与甲进行比赛），因此，图中显示的比赛总场次为：

$$5 \times 5-5=20$$

可是，看图 4-3 还可发现一个问题，每位选手进行了两场比赛。例如，从行方向来看，第 1 行中，甲与乙进行了一场比赛，在第 2 行中乙与甲也进行了一场比赛。如果甲与乙进行比赛、乙再与甲进行比赛，甲、乙两位选手就进行了两场比赛。其他选手的比赛场次也与此相同。

这样，就出现了计数中的“重复”问题，即某一现象被重复计数。

根据比赛规则，每一位选手只能与其余的每位选手进行一场比赛。因此，选手对阵表应如图 4-4 所示，这时只考虑右上角那些打对勾的比赛场次就行了。

	甲	乙	丙	丁	戊
甲	×	√	√	√	√
乙	○	×	√	√	√
丙	○	○	×	√	√
丁	○	○	○	×	√
戊	○	○	○	○	×

图 4-4

从图 4-4 中可看到各选手与对手的比赛场次：

甲：4场（与乙、丙、丁、戊比赛）

乙：3场（与丙、丁、戊比赛）

丙：2场（与丁、戊比赛）

丁：1场（与戊比赛）

戊：0场

即，比赛的总场次为：

$$4+3+2+1$$

$$=(1+4)\times 4\div 2$$

$$=10$$

经过 10 场比赛，各选手都与对手进行了一次单循环比赛，没有遗漏，也没有重复。

从上面的算式中提取相应的规律，对于有 n 位选手参加的单循环比赛，需要举行的比赛总场次为：

$$(n-1)+(n-2)+\cdots+3+2+1$$

$$=[(n-1)+1]\times(n-1)\div 2$$

$$=[n\times(n-1)]\div 2$$

例如，有 5 位选手参赛，则比赛的总场次为：

$$(5\times 4)\div 2=10$$

若有 10 位选手参赛，则比赛的总场次为：

$$(10\times 9)\div 2=45$$

4.2 乘法原理

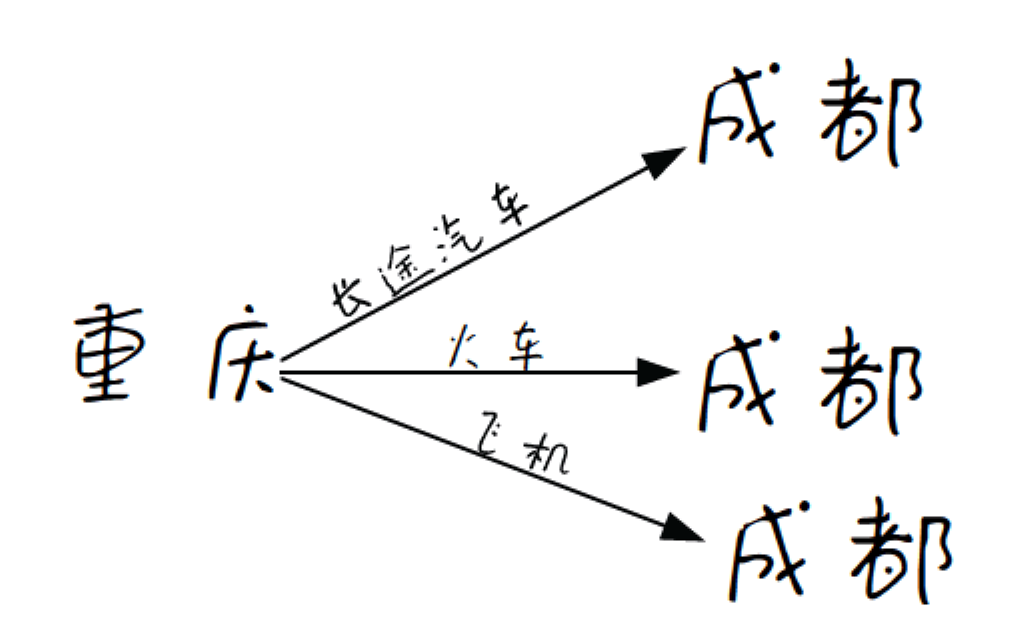
在实际应用中，要将所有情况都列出来，经常要用到乘法原理和加法原理。下面先来看看乘法原理的应用。

4.2.1 行程安排的问题

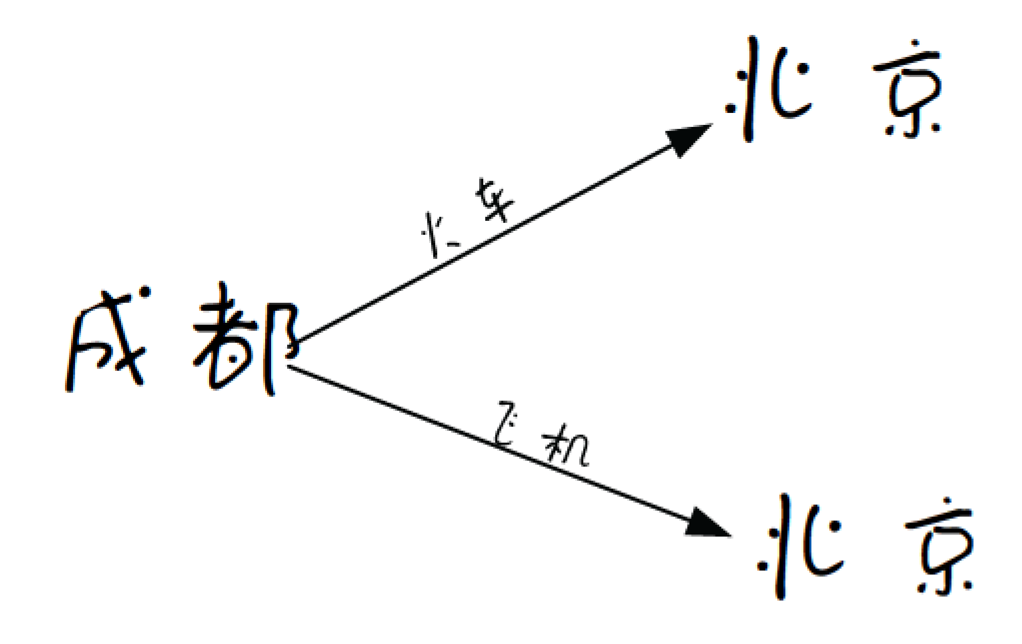
在日常生活中常常会遇到这样一些问题，就是在做一件事时，要分几步才能完成，而在完成每一步时，又有几种不同的方法。要知道完成这件事一共有多少种方法，就需要用乘法原理来解决。

例如：某公司销售部王经理从重庆到成都参加西南片区的销售会议，之后再到北京参加全国销售会议。其中，他从重庆到成都可以乘长途汽车、火车或飞机，从成都到北京可以乘火车或飞机。那么，王经理从重庆经成都到北京共有多少种不同的走法？

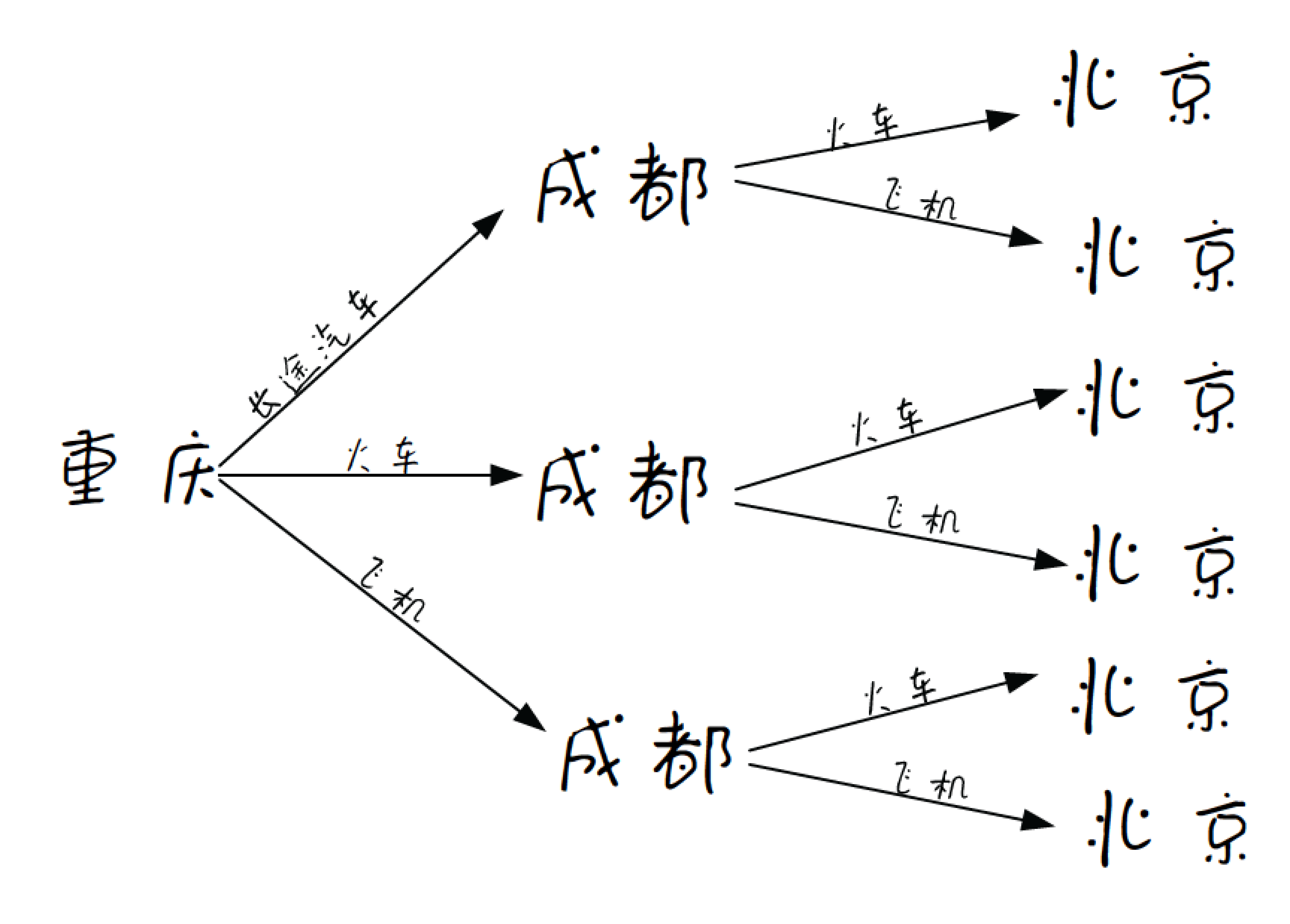
分析这个问题发现，王经理从重庆到北京要分两步走，第一步是从重庆到成都，可以有 3 种走法，即：



第二步是从成都到北京，有两种走法，即：



所以，王经理从重庆经成都到北京共有下面的 6 种走法：

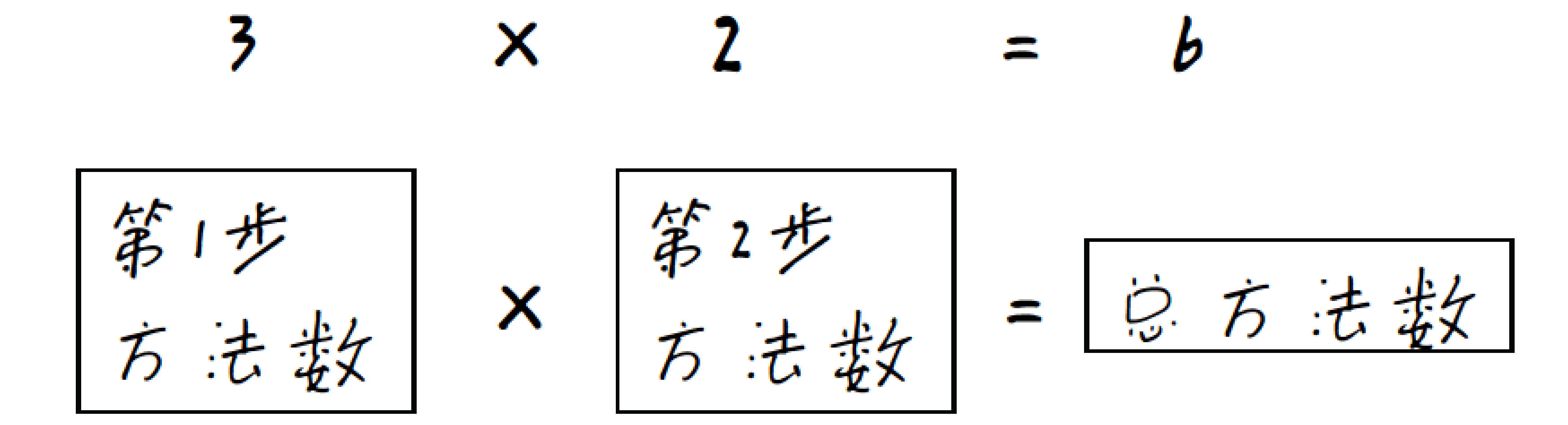


在上面讨论问题的过程中，我把所有可能的办法一一列举出来，这种方法叫穷举法。穷举法对于讨论方法数不太多的问题是很有效的，但是如果每一步中的方法数很多时，需要重复列出很多项目，费时费力，还容易出错。例如，在上例中，如果从重庆到成都有 10 种（或 20 种走法），从成都到北京又有 10 种（或更多）的走法，要将每种走法列举出来，将是比较繁琐的事。

4.2.2　乘法原理适用条件

为了解决穷举法比较繁琐的这个问题，可以对上例的行程问题做一番总结，从中提出相应的规律，以方便解决类似问题。

上面的例子中，完成一件事要分两个步骤，由穷举法得到的结论可看出，用第 1 步所有的可能方法数乘以第 2 步所有的可能方法数，就是完成这件事的总方法数。



一般地，如果完成一件事需要 n 个步骤，其中，做第 1 步有 M_1 种不同的方法，做第 2 步有 M_2 种不同的方法，做第 3 步有 M_3 种不同的方法……，做第 n 步有 M_n 种不同的方法，那么，完成这件事一共有 N 种方法，由各步的方法数相乘而得到。

$$N=M_1 \times M_2 \times M_3 \times \cdots \times M_n$$

这就是乘法原理。

哪些情况下适用乘法原理？

乘法原理的核心就是分步：每步都只完成其中的一部分，只有每一步都完成了这件事才算完成。分步计数时应注意步与步之间的连续性和独立性，以确保在计数时不遗漏不重复。

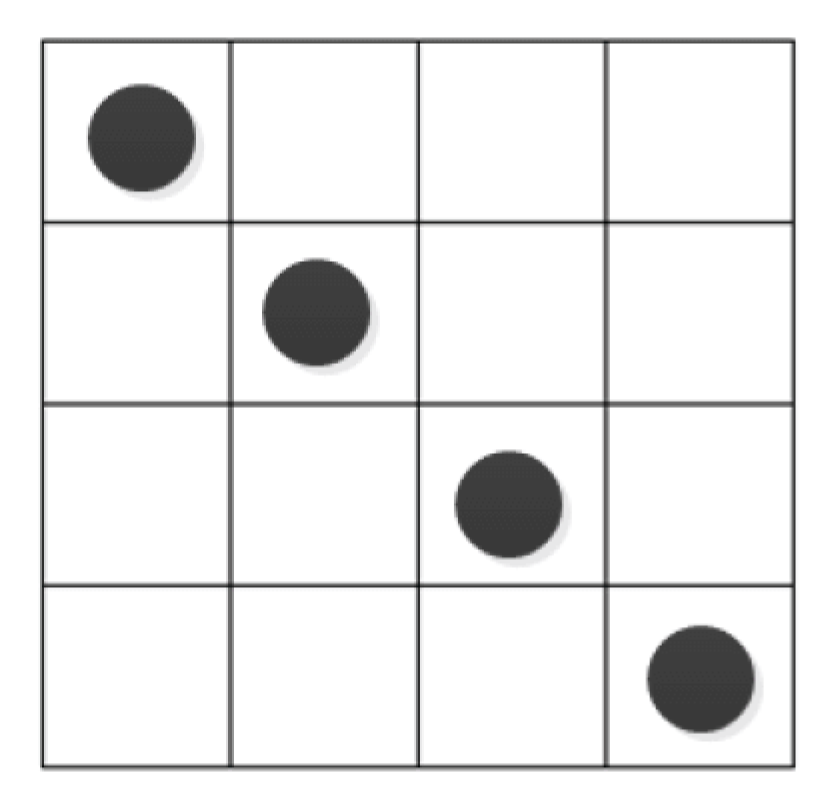
因此分步完成的任务可适用乘法原理，有些问题可能需要仔细分解一下才能化解为分步完成的结构。

4.2.3 棋盘上棋子的放法

再来看一个例子。如图 4-5 所示是一个 4×4 方格的棋盘，要在该棋盘中放 4 枚棋子，使每行每列只能出现一个棋子，问：共有多少种不同的放法（图 4-5 是其中一种放置棋子的方法）？

由于 4 枚棋子要一个一个地放入棋盘的方格内，因此可看成是分 4 步解决这个问题。

（1）放置第 1 枚棋子。此时由于 16 个棋盘方格都没有棋子，因此，这枚棋子可放在任意一个方格中，有 16 种不同的放法，如图 4-6 所示。


图 4-5

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

图 4-6

（2）放置第 2 枚棋子。由于棋盘中已经放置了 1 枚棋子，放第 1 枚棋子那一行和一列中的其他方格内也不能放第 2 枚棋子，因此还剩下 9 个方格可以放第 2 枚棋子，即第 2 枚棋子有 9 种放法，如图 4-7 所示。

①			
	1	2	3
	4	5	6
	7	8	9

1		2	3
	①		
4		5	6
7		8	9

图 4-7

（3）放置第 3 枚棋子。这时需去掉第 1、2 枚棋子所在的行和列的方格，还剩下 4 个方格可以放置第 3 枚棋子，则第 3 枚棋子有 4 种放法，如图 4-8 所示。

①			
	②		
		1	2
		3	4

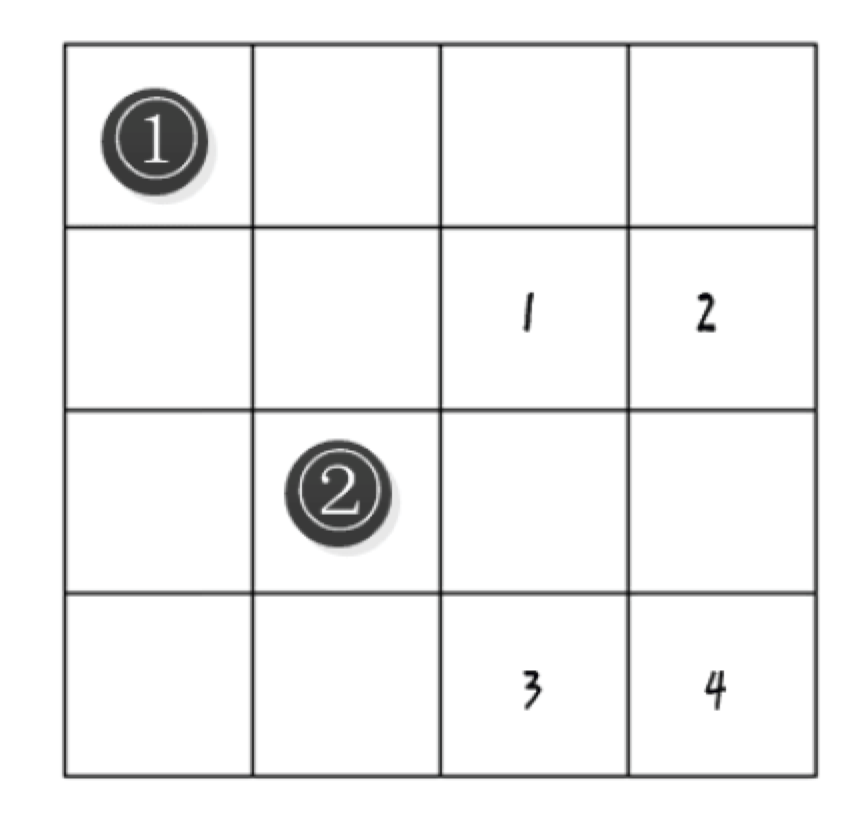


图 4-8

（4）放第 4 枚棋子。由于要去掉前 3 枚棋子所在行和列的方格，只剩下 1 个方格可以放第 4 枚棋子了，因此第 4 枚棋子只有 1 种放法，如图 4-9 所示。

①			
	②		
		③	
			1

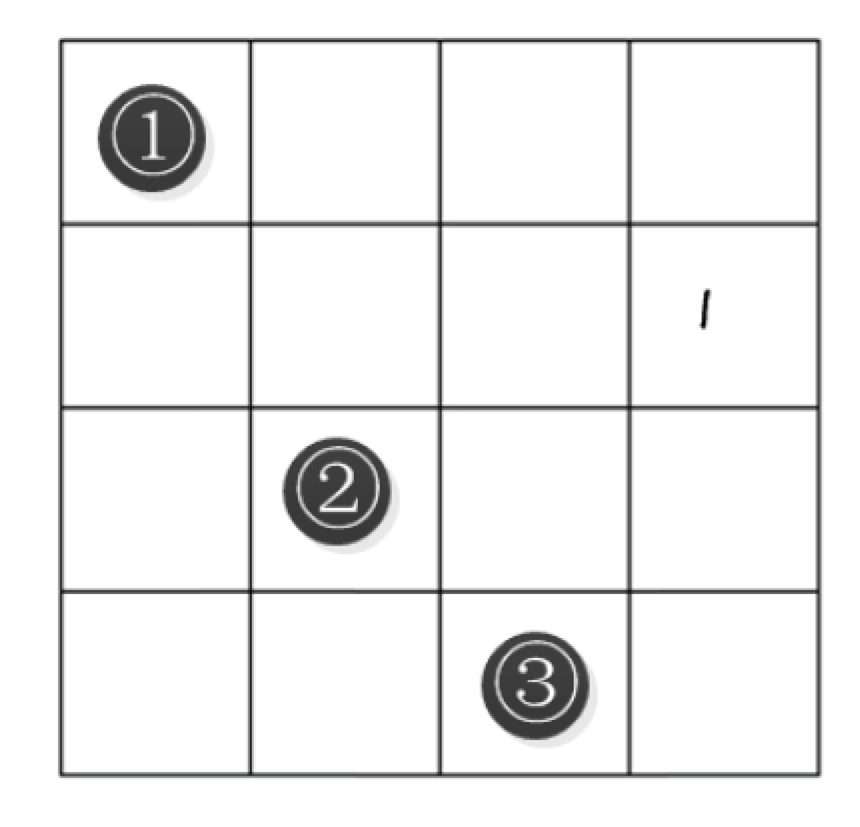


图 4-9

根据上面的分析，用乘法原理即可求出 4 枚棋子的放置方法数量，共有：

$$16 \times 9 \times 4 \times 1 = 576$$

种不同的放法。

4.2.4 买彩票保证中奖的方法

接下来说一个轻松的话题，说说用乘法原理来计算彩票的问题。

目前我国有福利彩票和体育彩票两大类，为国家的福利和体育事业做出了贡献。民众购买彩票为相关事业做出贡献，也有可能中奖改善自己的生活。

作为购买彩票的民众来说，肯定是希望自己能中大奖，那么，该怎么买彩票才能确保中奖呢？

由于彩票中奖号码是由摇奖机随机生成的，没办法猜中，因此，要确保中奖，只有将所有彩票号码全部买完没有遗漏，才可确保中奖！

那么，所有彩票号码共有多少种可能？要花多少钱才能买完？这就需要我们用乘法原理将所有可能都计算出来。

例如，对于体育彩票中的“七星彩”，其游戏规则是：彩票号码长度为 7 位，每位数字为 0～9 中的一个。那么，七星彩的彩票号码有多少种可能？

如图 4-10 所示，绘制 7 个方框用来填写彩票号码。

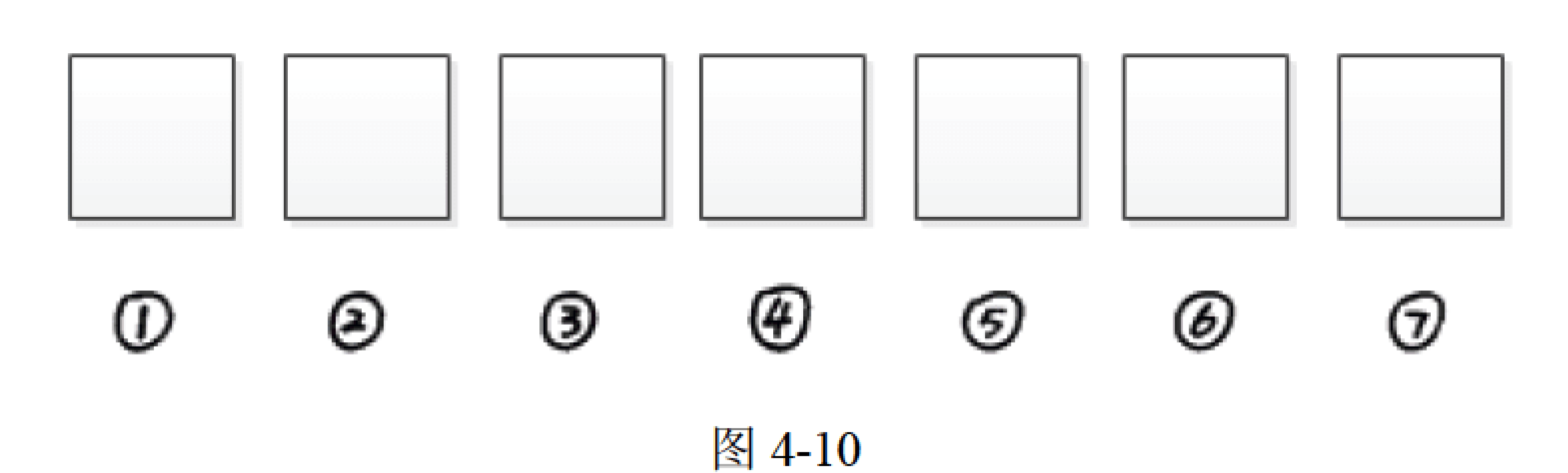


图 4-10

由于有 7 个号码，可以认为是分 7 个步骤来生成彩票号码。

（1）填第 1 位号码。可以有 0～9 这 10 个数字可用，共 10 种可能。

（2）填第 2 位号码。可以有 0～9 这 10 个数字可用，共 10 种可能。

（3）填第 3 位号码。可以有 0～9 这 10 个数字可用，共 10 种可能。

（4）填第 4 位号码。可以有 0～9 这 10 个数字可用，共 10 种可能。

（5）填第 5 位号码。可以有 0～9 这 10 个数字可用，共 10 种可能。

（6）填第 6 位号码。可以有 0～9 这 10 个数字可用，共 10 种可能。

（7）填第 7 位号码。可以有 0～9 这 10 个数字可用，共 10 种可能。

根据乘法原理，七星彩的号码总数共有：

① ② ③ ④ ⑤ ⑥ ⑦

$$10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 = 10\ 000\ 000$$

对于七星彩来说，由于各位数字允许相同，因此计算起来很简单。全部七星彩号码有 1000 万个，将这 1000 万个号码全部买完可保证中特等奖。

4.3　加 法 原 理

前面曾提到，要将所有情况都列出来，还经常要用到加法原理。下面就来看看加法原理的特点和应用。

4.3.1　仍然是行程问题

首先，仍然来看行程问题。

日常生活中常有这样的情况，就是在做一件事时，有几类不同的方法，而每一类方法中，又有几种可能的做法。那么，考虑完成这件事所有可能的做法，就要用我们将讨论的加法原理来解决。

例如，某公司销售经理王经理从重庆到成都参加会议，他可以乘火车也可以乘长途汽车，现在知道每天有 13 趟火车从重庆到成都，有 8 趟长途汽车从重庆到成都。那么王经理在一天中去成都能有多少种不同的走法？

分析这个问题发现，王经理从重庆去成都要么乘火车，要么乘长途汽车，有这两大类走法，如果乘火车，有 13 趟火车可选，如果乘长途汽车，有 8 趟汽车可选。

重庆 —每天13趟火车→ 成都

重庆 —每天8趟汽车→ 成都

那么，王经理在一天中可以有 13+8=21 种不同的走法。

13+8=21

4.3.2　总结出的加法原理

在上面的问题中，完成一件事有两大类不同的方法。在具体做的时候，只要采用一类中的一种方法就可以完成，并且两大类方法是互无影响的，那么完成这件事的全部做法的总数，就是用第一类的方法数加上第二类的方法数，这就是加法原理。

一般地，如果完成一件事有 N 类方法，第一类方法中有 M_1 种不同做法，第二类方法中有 M_2 种不同做法，第三类方法中有 M_3 种不同做法……，第 N 类方法中有 M_n 种不同的做法，则完成这件事共有：

$$N=M_1+M_2+M_3+\cdots+M_n$$

种不同的方法。

选用加法原理的条件：如果完成一件事中有 N 类方法，这 N 类方法彼此之间是相互独立的，无论用哪一类方法中的某种方法都能单独完成这件事，求完成这件事的方法种类，则可使用加法原理。

在加法原理中的分类，是指一件事在一定标准下进行的分类，分类标准不同，得到的分类也不同，分类要满足不重复不遗漏的要求。完成这件事的各种方法是相互独立、相互排斥的，每一种方法都能完成这件事。

结合前面用的乘法原理，要区分加法原理与乘法原理，只需要注意，如果做一件事，完成它若是有 N 类办法，是分类问题，每一类中的方法都是独立的，因此使用加法原理；做一件事，需要分 N 个步骤，步与步之间是连续的，只有将分成的若干个互相联系的步骤，依次相继完成，这件事才算完成，就适用乘法原理。

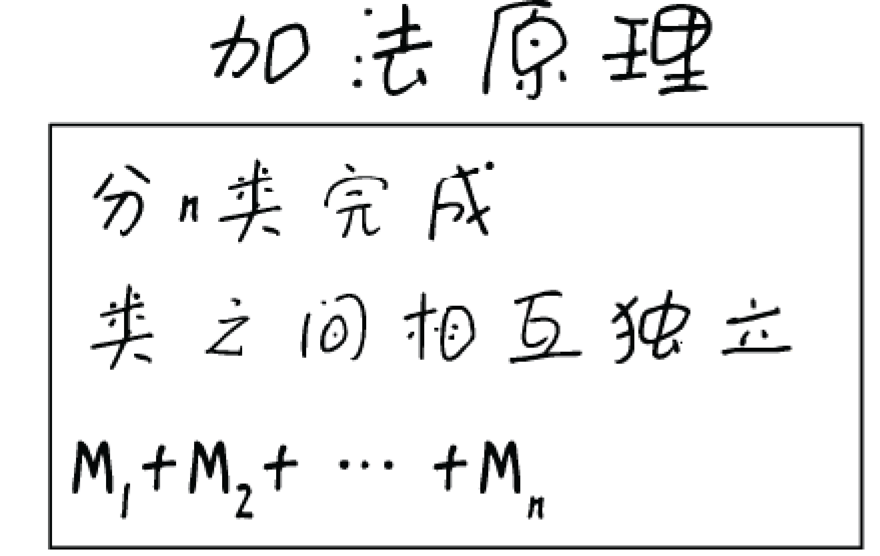
完成一件事的分“类”和“步”是有本质区别的，因此也就将两个原理区分开了。

乘法原理

分n步完成

步之间相互关联

$M_1\times M_2\times\cdots\times M_n$



4.3.3　骰子出现偶数的次数

在很多游戏中，都需要用到骰子中的点数来决定游戏的进程。一粒骰子共有 6 面，分别标出 1、2、3、4、5、6 这 6 个数字。如果将两粒骰子同时掷下，向上一面的数字之和为偶数的有多少种情形？如图 4-11 所示，两粒骰子的点数分别为 5 和 3，加起来总点数为 8，则说明两粒骰子总点数为偶数。

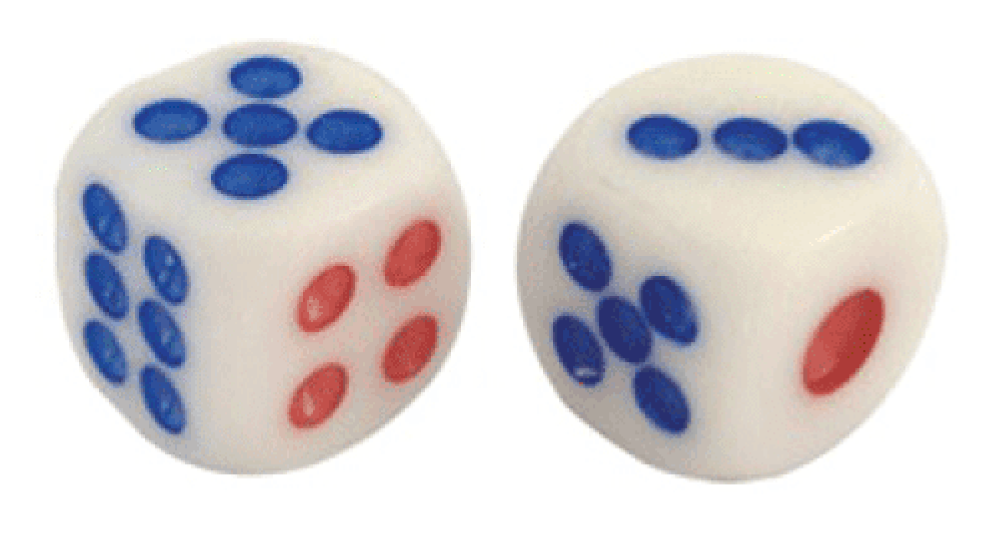


图 4-11

要使两粒骰子点数之和为偶数，只要这两粒骰子点数中数字的奇偶性相同即可。即

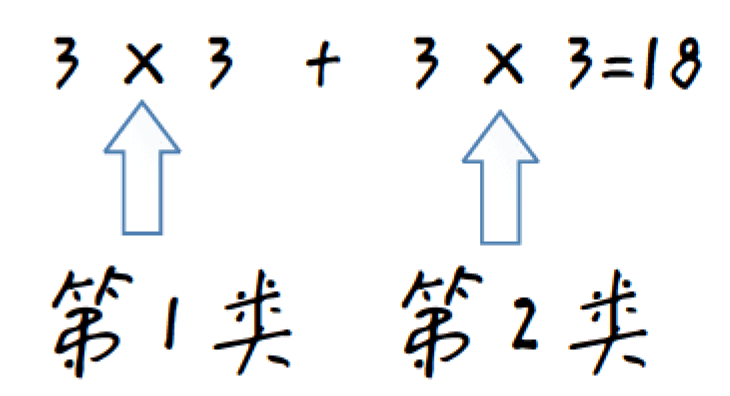
这两个骰子中的数字要么同为奇数，要么同为偶数，若为一奇一偶，相加的和必为奇数。所以，可分两大类来考虑（注意是分为两大类，不是两步）。

第一类，两个数字同为奇数。由于两粒骰子可以分成两步来查看其点数（这里是分步，不是分类），查看第 1 粒骰子时，出现奇数有 3 种可能，即 1、3、5；查看第 2 粒骰子时，出现奇数也有 3 种可能。根据乘法原理，这时共有 3×3=9 种不同的情形。

第二类，两个数字同为偶数，类似第一类的讨论方法，也有 3×3=9 种不同情形。

	第1粒骰子	第2粒骰子
奇数	3种可能(1,3,5)	3种可能(1,3,5)
偶数	3种可能(2,4,6)	3种可能(2,4,6)

根据以上的分析可看出，在求解骰子出现偶数的可能性时，既有分“类”也有分“布”，也就是说，需要将乘法原理和加法原理结合在一起使用。首先用乘法原理求出两粒骰子同为奇数，使两粒骰子总点数为偶数的情况，再用乘法原理求出两粒骰子同为偶数，使两粒骰子总点数为偶数的情况。最后用加法原理将这两类方法的数量相加。



即，两粒骰子点数总和为偶数的情形共有 18 种。

4.4　排列与组合的关系

在本章前面学习了通过计数将所有情况都列出来的方法。下面开始研究排列与组合。根据本章开篇给出的定义，排列是从给定个数的元素中取出指定个数的元素进行排序。组合则是指从给定个数的元素中仅仅取出指定个数的元素，不考虑排序。可以看出，排列与组合的区别就是看问题是否和顺序有关，有关就是排列，无关就是组合。

4.4.1　排列

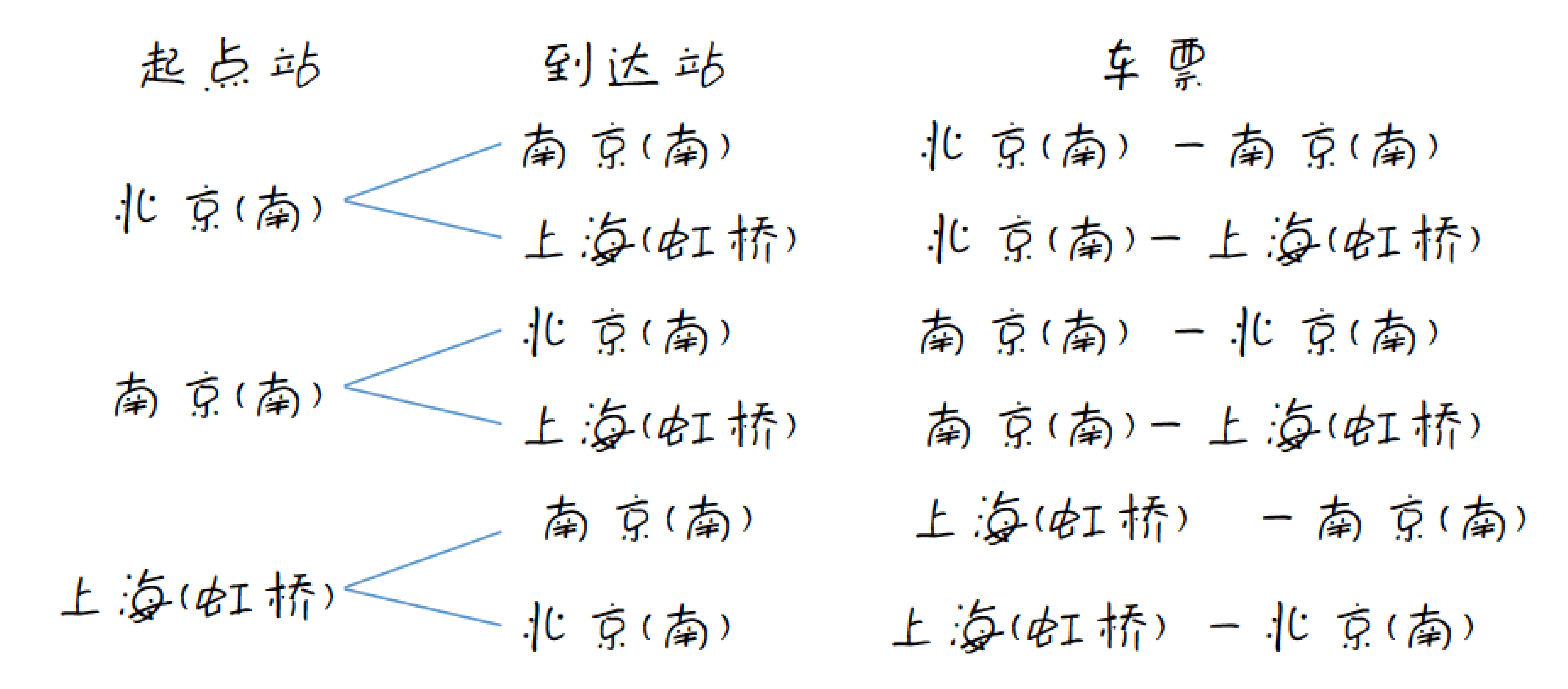
在实际生活中常遇到这样的问题，就是要把一些事物排在一起，构成一列，计算有多少种排法，这就是排列问题。

在排列的过程中，结果不仅与参加排列的元素有关，而且与各元素所在的先后顺序也有很重要的关系。

1. 有多少种车票？

例如：在北京（南）与上海（虹桥）之间的高速动车 G3/G4，途经南京（南），开通这条线路的高速动车共需要准许多少种不同的车票？

分析这个问题，可以用枚举法解决，3 个城市之间的火车票有下面 6 种方式：

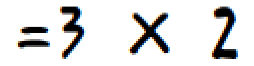


如果不用枚举法，注意到要准备的火车票的种类不仅与所选的两个城市有关，而且与这两个城市作为起点站、到达站的顺序有关，所以，要考虑共准备多少种不同的火车票，就要在 3 个城市之间每次取出两个，按照起点站、到达站的顺序排列。即，需要分为两步来进行操作：

（1）确定起点站，在 3 个城市中，任取一个为起点站，共有 3 种选法。

（2）确定到达站，每次确定了 1 个起点站后，只能从剩下的两个城市之中选出到达站，因此就只有两种选法。

对于这种分步完成的任务，根据乘法原理，可计算出需要准备的火车票种类：



$$=6$$

为叙述方便，我们把研究对象（如上例中的火车站点）看作为元素，那么上面的问题就是在 3 个不同的元素中取出两个，按照一定的顺序排成一列的问题。

我们把每一种排法叫做一个排列（如“北京（南）——南京（南）”就是一个排列），把所有排列的个数叫做排列数。

那么上例中求火车票种类数量的问题就是求排列数的问题。可以看出，求排列数的问题，可以通过乘法原理进行快速计算。

既然知道了可以通过乘法原理来解决这个问题，那么对更复杂的情况也可以方便地解决了。例如，已知从北京（西）到广州（南）这条高速动车 G7/9 一共有 6 个站，那么这列高速动车共需要准备多少种不同的车票？

如果用枚举法，则可能需要写出很长一串站名了，先将 6 个站分别作为起点站，再将剩下的 5 个站作为到达站。显然，这种方式太繁琐了。

根据我们前面推导出的规律，可以用乘法原理解决这个问题，分两步，第 1 步在 6

个站点中取 1 个作为起点站（有 6 种可能）。第 2 步，在剩下的 5 个站点中选一个作为到达站（有 5 种可能），因此需准备的车票种类数量有：

$$=6\times5$$

$$=30$$

2. 旗语

由于火车票中只有起点站和到达站两个数据，只需要两个步骤即可完成，根据乘法原理，只需要将这两步中可选择站点的数量相乘即可得到车票种类数量。在实际生活中还经常遇到更复杂的情况。

我们来研究一个更复杂的例子：旗语。旗语在古代是一种主要的通信方式，现在是世界各国海军通用的语言。不同的旗子，不同的旗组表达着不同的意思。现假设两只船之间约定用 5 面颜色各异（由红、黄、蓝、白、绿）的旗帜来传递信息，将这 5 面旗帜放在船头固定位置。当这种 5 面旗帜按不同颜色排列时分别表达不同的信息，则这两只船之间通过 5 色旗能传递多少种含义不同的信息？

分析：用字母 R 表示红色、Y 表示黄色、B 表示蓝色、W 表示白色、G 表示绿色，则 5 种颜色的旗子可排列成如图 4-12 所示的不同形式。

R Y B W G	R Y B G W	R Y W B G	R Y W G B	R Y G B W	R Y G W B
R B Y W G	R B Y G W	R B W Y G	R B W G Y	R B G Y W	R B G W Y
R W Y B G	R W Y G B	R W B Y G	R W B G Y	R W G Y B	R W G B Y
R G Y B W	R G Y W B	R G B Y W	R G B W Y	R G W Y B	R G W B Y
Y R B W G	Y R B G W	Y R W G B	Y R W B G	Y R G B W	Y R G W B
Y B R W G	Y B R G W	Y B W R G	Y B W G R	Y B G R W	Y B G W R
Y W R B G	Y W R G B	Y W B R G	Y W B G R	Y W G R B	Y W G B R
Y G R B W	Y G R W B	Y G B R W	Y G B W R	Y G W R B	Y G W B R
B R Y W G	B R B G W	B R W Y G	B R W G Y	B R G B W	B R G W B
B Y R W G	B Y R G W	B Y W R G	B Y W G R	B Y G R W	B Y G W R
B W R Y G	B W Y G Y	B W Y R G	B W Y G R	B W G R Y	B W G Y R
B G R Y W	B G R W Y	B G Y R W	B G Y W R	B G W R Y	B G W Y R
W R Y B G	W R Y G B	W R B Y G	W R B G Y	W R G Y B	W R G B Y
W Y R B G	W Y R G B	W Y B R G	W Y B G R	W Y G R B	W Y G B R
W B R Y G	W B R G Y	W B Y R G	W B Y G R	W B G R Y	W B G Y R
W G R Y B	W G R B Y	W G Y R B	W G Y B R	W G B R Y	W G B Y R
G R Y B W	G R Y W B	G R B Y W	G R B W Y	G R W Y B	G R W B Y
G Y R B W	G Y R W B	G Y B R W	G Y B W R	G Y W R B	G Y W B R
G B R Y W	G B R W Y	G B Y R W	G B Y W R	G B W R Y	G B W Y R
G W R Y B	G W R B Y	G W Y R B	G W Y B R	G W B R Y	G W B Y R

图 4-12

图 4-12 中的排列是按以下方法来处理的（共分 5 步，分别设置 5 种颜色旗帜的放置位置）：

（1）从 5 种颜色的旗帜中取 1 面旗放在第 1 个位置，有 5 种取法。

（2）从剩下的 4 种颜色的旗帜中取 1 面旗放在第 2 个位置，有 4 种取法。

（3）从剩下的 3 种颜色的旗帜中取 1 面旗放在第 3 个位置，有 3 种取法。

（4）从剩下的 2 种颜色的旗帜中取 1 面旗放在第 4 个位置，有 2 种取法。

（5）只剩下 1 种颜色的旗帜，直接放在第 5 个位置，只有 1 种取法。

根据乘法原理可得：

$$=5\times 4\times 3\times 2\times 1$$

$$=120$$

即，5 色旗有 120 种排列，可表示 120 种不同的信息。

3. 排列的数学表示

根据上面的例子，对排列进行归纳总结，可得到以下规律：

一般地，从 n 个不同的元素中任取出 m 个（$m \leqslant n$）元素，按照一定的顺序排成一列，叫做从 n 个不同元素中取出 m 个元素的一个排列。

由排列的定义可以看出，如果两个排列相同，不仅表示这两个排列中的元素完全相同，而且各元素的先后顺序也一样。如果两个排列的元素不完全相同，或者各元素的排列顺序不完全一样，则表示是两个不同的排列。

从 n 个不同元素中取出 m 个（$m \leqslant n$）元素的所有排列的个数，叫做从 n 个元素中取出 m 个元素的排列数，在数学里将其记为：

$$P_n^m$$

那么，上面这个排列的表示该怎么计算呢？

根据前面的两个例子可以归纳出如下计算步骤：

（1）先排第一个位置上的元素，可以从 n 个元素中任选一个，有 n 种不同的选法。

（2）排第 2 个位置上的元素。这时，由于第一个位置已用去了 1 个元素，只剩下（n-1）个不同的元素可供选择，共有（n-1）种不同的选法。

（3）排第 3 个位置上的元素，有（n-2）种不同的选法；

……

第 m 步：排第 m 个位置上的元素。由于前面已经排了（m-1）个位置，用去了（m-1）个元素。这样，第 m 个位置上只能从剩下的[n-（m-1）]=（n-m+1）个元素中选择，有（n-m+1）种不同的选法。

因此，根据乘法原理，可列出排列数的计算公式如下：

$$P_n^m = n \times (n-1) \times (n-2) \times \dots \times (\mathrm{n}-\mathrm{m}+1)$$

这里，$m \leqslant n$，且等号右边从 n 开始，后面每个因数比前一个因数小 1，共有 m 个因数相乘。

例如，对前面关于旗语的例子进行修改，因为用 5 面 5 色旗可以表示 120 种状态，含义太多，旗手也可能记不住这么多状态的含义，也用不了这么多种状态。现在进行精简，仍然使用 5 色旗，但是每次只用 3 面旗帜，求这种方式的旗帜组合能表示出多少种不同的含义？

根据前面总结的规律，可用以下算式来进行计算：

$$P_5^3 = 5 \times 4 \times 3$$

$$= 60$$

如果从 5 色旗中选 4 面旗，可以表示多少种不同的含义呢？

$$P_5^4 = 5 \times 4 \times 3 \times 2$$

$$= 120$$

可以看出，将 5 面旗帜全部用上和只用 4 面旗帜，得到的排列数完全相同。这是由于，把前 4 面旗帜选定之后，就只剩最后那一面旗帜，也就没办法选择了。因此，最后一面旗帜用不用已经无所谓了。

从排列的计算公式也可看出，若从 n 个元素中选择 n 个元素进行排列，可得到以下计算公式：

$$\begin{aligned} P_n^n &= n \times (n-1) \times (n-2) \times \dots \times (\mathrm{n}-\mathrm{n}+1) \\ &= n \times (n-1) \times (n-2) \times \dots \times 2 \times 1 \end{aligned}$$

可以看出，这就相当于是计算 n 的阶乘了。

若从 n 个元素中选择 n-1 个元素进行排列，可得到以下计算公式：

$$\begin{aligned} P_n^{n-1} &= n \times (n-1) \times (n-2) \times \dots \times [n-(n-1)+1] \\ &= n \times (n-1) \times (n-2) \times \dots \times 2 \end{aligned}$$

可以看出：

$$P_n^{n-1} = P_n^n$$

所以，在上面旗语的例子中，从 5 色旗中选择 4 面旗帜就可以表示出最多的状态数量了。

4.4.2 组合

日常生活中有很多“分组”问题，例如，在体育比赛中，把参赛队分为几个组（这时分在同一组中的队友之间的位置顺序并不重要），这种“分组”问题，就是我们下面将要讨论的组合问题，这里，我们将着重研究有多少种分组方法的问题。

1．有多少种票价？

仍然来看火车票的问题。

在前面计算排列数的例子中，我们已经知道在北京（南）与上海（虹桥）之间的高速动车 G3/G4，途经南京（南），开通这条线路的高速动车会有 6 种不同车票。现在要求这些火车票共有几种价格（假设相同两站之间往返票价相同）？

由于往返票价格相同，这时，从“北京（南）”到“南京（南）”的车票价格与从“南京（南）”到“北京（南）”的票价相同。

北京（南）－南京（南）　票价748.50元
南京（南）－北京（南）　票价748.50元

因此，求有多少种价格的问题实际上就是计算从 3 个城市中取两个城市，有多少种不同的取法，即这时只与考虑的两个城市有关而与两个城市的顺序无关。

通过枚举法可求出有以下 3 种票价：

北京（南）⟷ 南京（南）
北京（南）⟷ 上海（虹桥）
南京（南）⟷ 上海（虹桥）

这是通过枚举法推导出来的数量，如果要计算的车站数量增多，用这种枚举方法显然就不方便了。

那么，有没有办法通过公式进行计算呢？

我们来推导一下。要计算车票价格的种类可按以下方式考虑：

（1）仍然按“排列”的方式计算出有多少种排列数。

（2）剔除重复计数的部分。

到这里，重点就是该怎么计算重复部分了。

在上面计算车票价格的问题中，每张车票需要关注的有两个站点名称，A 站到 B 站

和 B 站到 A 站的票价是一样的，但在第（1）步的排列中却计算了 2 次，实际车票价格只有一种。所以，重复计数 1 次。这样，只需要将第（1）步计算的排列数除以 2，就可以得到不同的车票价格数量了。

根据以上推导过程，如果一列火车有 10 个站点，则这列火车的票价种数为：

$$P_{10}^{2} \div 2$$

$$=10 \times 9 \div 2$$

$$=45$$

2．有多少组旗语？

在上面旗语的例子中，如果不关注各色旗帜的排放顺序，5 色旗能有多少种组合呢？

由于不区分各色旗帜的排列顺序，因此，如下面的 5 种排列其实都只能算是一种组合，因为都是由红（R）、黄（Y）、蓝（B）、白（W）、绿（G）这 5 种颜色的旗帜组成的。

R	Y	B	W	G		R	Y	B	G	W		R	Y	W	B	G		R	Y	W	G	B		R	Y	G	B	W		R	Y	G	W	B

从图 4-12 中可看出，图中的 120 种排列情况中的每种排列都包含 5 种颜色的旗帜，因此，若用 5 面 5 色旗帜只能构成一种组合。

那么，若只取 5 色旗中的 4 面旗帜，可以构成多少种组合呢？

根据我们上面推导出的火车票价格种类的方法，同样可以分成两步来计算旗语的组合种类，即：

（1）按“排列”的方式计算出旗语有多少种排列数。

（2）剔除重复计数的部分。

对于第（1）步，计算从 5 色旗中取 4 面旗可构成多少排列数，可按以下公式计算：

$$P_{5}^{4}=5 \times 4 \times 3 \times 2$$

$$=120$$

在第（2）步，需要计算重复计数的部分，将这部分从排列数中剔除，才能得到实际的组合种类数。该如何计算出重复的数据呢？

在这个例子中要计算出重复数，不如计算不同火车票价种类好理解，因为火车票价只需要 2 个元素，而现在的例子中每一个组合要用 4 个元素（即 4 种颜色的旗帜）。

这时可以考虑用先选择 4 种颜色的旗帜，如选择红（R）、黄（Y）、蓝（B）、白（W），这 4 种颜色的旗帜按排列的方式可得到如图 4-13 所示的排列情况，有 24 种。

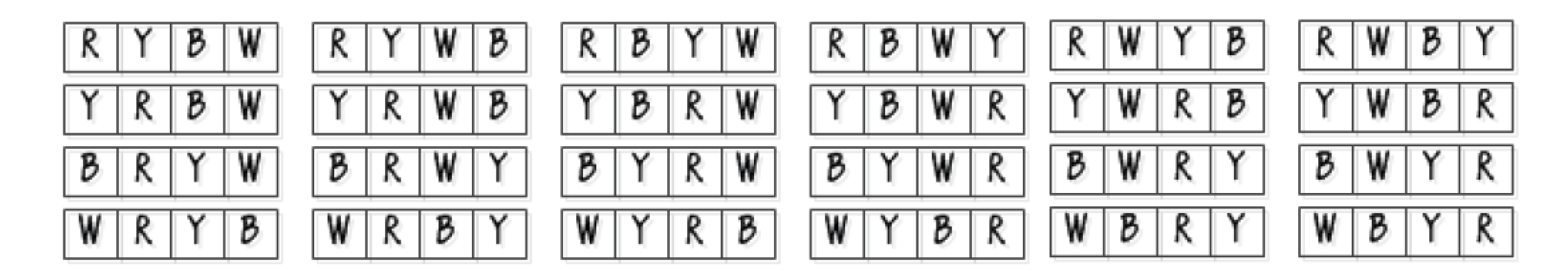


图 4-13

可以看出，图 4-13 是枚举的排列数，也可用以下公式计算：

$$P_4^4=4\times 3\times 2$$

$$=24$$

如图 4-13 所示，这 24 种排列其实只是一种组合。这样，就可将第（1）步中计算出的排列数 120，除以第（2）步中计算出的排列数 24，即可得到组合数 5。即从 5 色旗中选择 4 面旗帜可得到 5 种不同的组合。

3. 组合的数学表示

一般地，从 n 个不同元素中取出 m 个（$m\leq n$）元素组成一组不计较组内各元素次序的序列，叫做从 n 个不同元素中取出 m 个元素的一个组合。

由组合的定义可以看出，两个组合是否相同，只与这两个组合中的元素有关，而与取到这些元素的先后顺序无关，只有当两个组合中的元素不完全相同时，它们才是不同的组合。

从 n 个不同元素中取出 m 个元素（$m\leq n$）的所有组合的个数，叫做从 n 个不同元素中取出 m 个不同元素的组合数，记作：

$$C_n^m$$

根据前面的两个例子可得出组合数的计算公式：

$$C_n^m=P_n^m\div P_m^m$$

有了这个公式，就可以方便地计算出从 n 个元素中选择 m 个元素的组合数了。

例如，在计算不同火车票价格种类的例子，就是要计算从 3 个城市中取 2 个城市的组合数，可使用以下公式进行计算：

$$C_3^4=P_3^2\div P_2^2$$

$$=(3\times 2)\div(2\times 1)$$

$$=3$$

而在旗语的例子中，从 5 色旗中选择 4 面旗帜共可得到的组合数可用以下公式进行计算：

$$C_5^4=P_5^4 \div P_5^4$$
$$=(5\times4\times3\times2)\div(4\times3\times2\times1)$$
$$=5$$

从 5 色旗中选择 3 面旗帜，共可得到多少组合数呢？

$$C_5^3=P_5^3 \div P_4^3$$
$$=(5\times4\times3)\div(3\times2\times1)$$
$$=10$$

由此可见，在 *n* 个元素中选择 *m* 个元素进行组合，得到的组合数与 *m* 有关。*m* 越大组合数反而越小，若 *m*=*n*，则组合数为 1。

4.4.3　排列与组合的联系

下面再来看看排列与组合的联系吧。以从 5 色旗中选择 3 面旗帜为例，来分析排列与组合的关系。

首先来看，如果取 3 面旗帜，可得到如图 4-14 所示的排列。这种用所有元素进行的排列称为全排列。

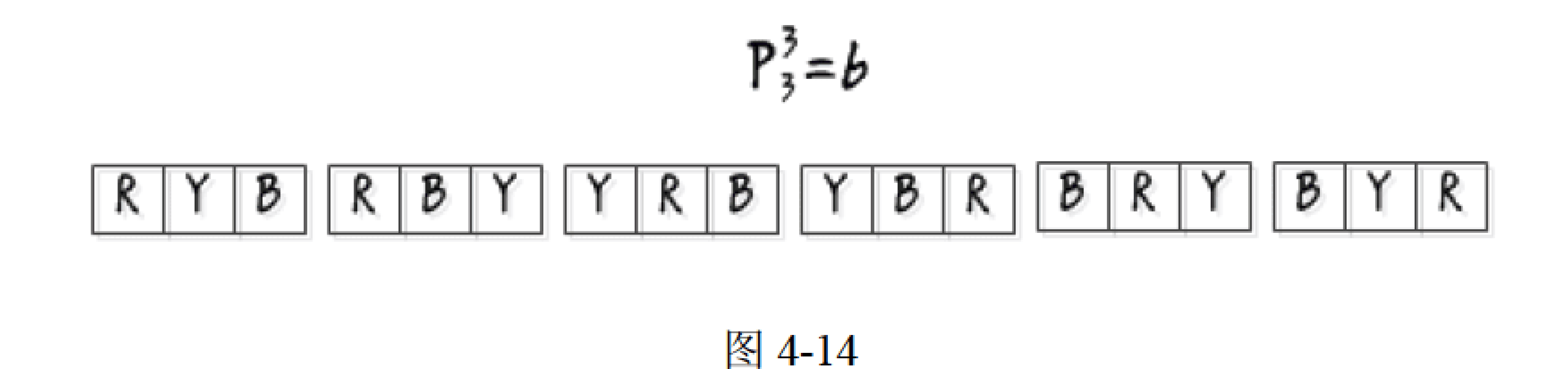


图 4-14

接着再来看从 5 色旗中取 3 面旗帜的组合情况，如图 4-15 所示。

将图 4-15 所示组合的第 1 个组合中的各色旗帜进行排列，就可得到图 4-14 所示的排列。若将其余各组合也进行类似的排列，就可得到如图 4-16 所示的排列效果（中间部分省略了）。

在图 4-16 中，已将排列数与组合数的关系通过算式列出来了，这个算式与上节介绍的组合数计算公式相符。

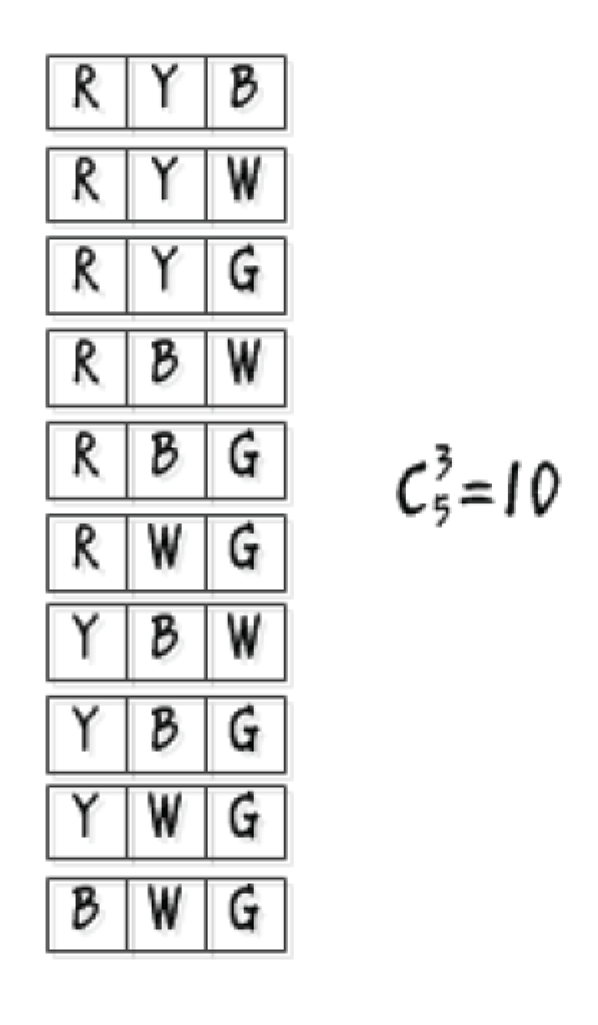


图 4-15

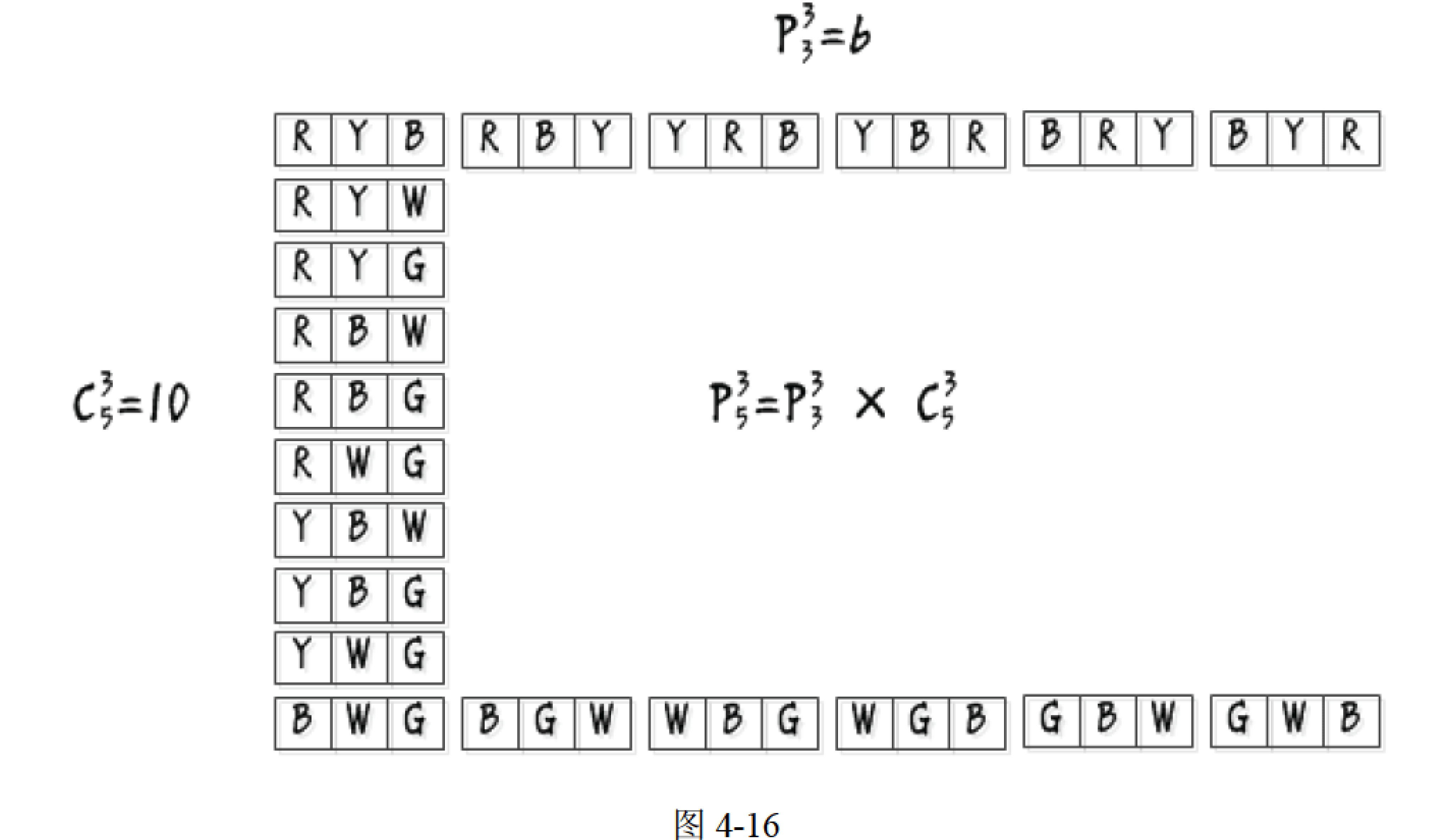


图 4-16

4.4.4　可重排列

在上面介绍的排列是从 n 个不同的元素中任取出 m 个（$m \leqslant n$）元素，按照一定的顺序排成一列。这里选取的元素数量 m 不能超过 n，并且当选取了一个元素之后，这个元素就不能再次被选取了。也就是说，在一个排列中，元素之间的值不会相同。

在现实生活中，还会出现另一种情况，就是在一个排列中，有的元素是相同的。例如，电话号码是由 0～9 这 10 个数字排列组成，各位数字有可能重复。如电话号码 82608833 中数字 8 出现了 3 次，数字 3 出现了 2 次。

对于这种从 n 个不同元素中取出 m 个元素（同一个元素允许重复取出），按照一定

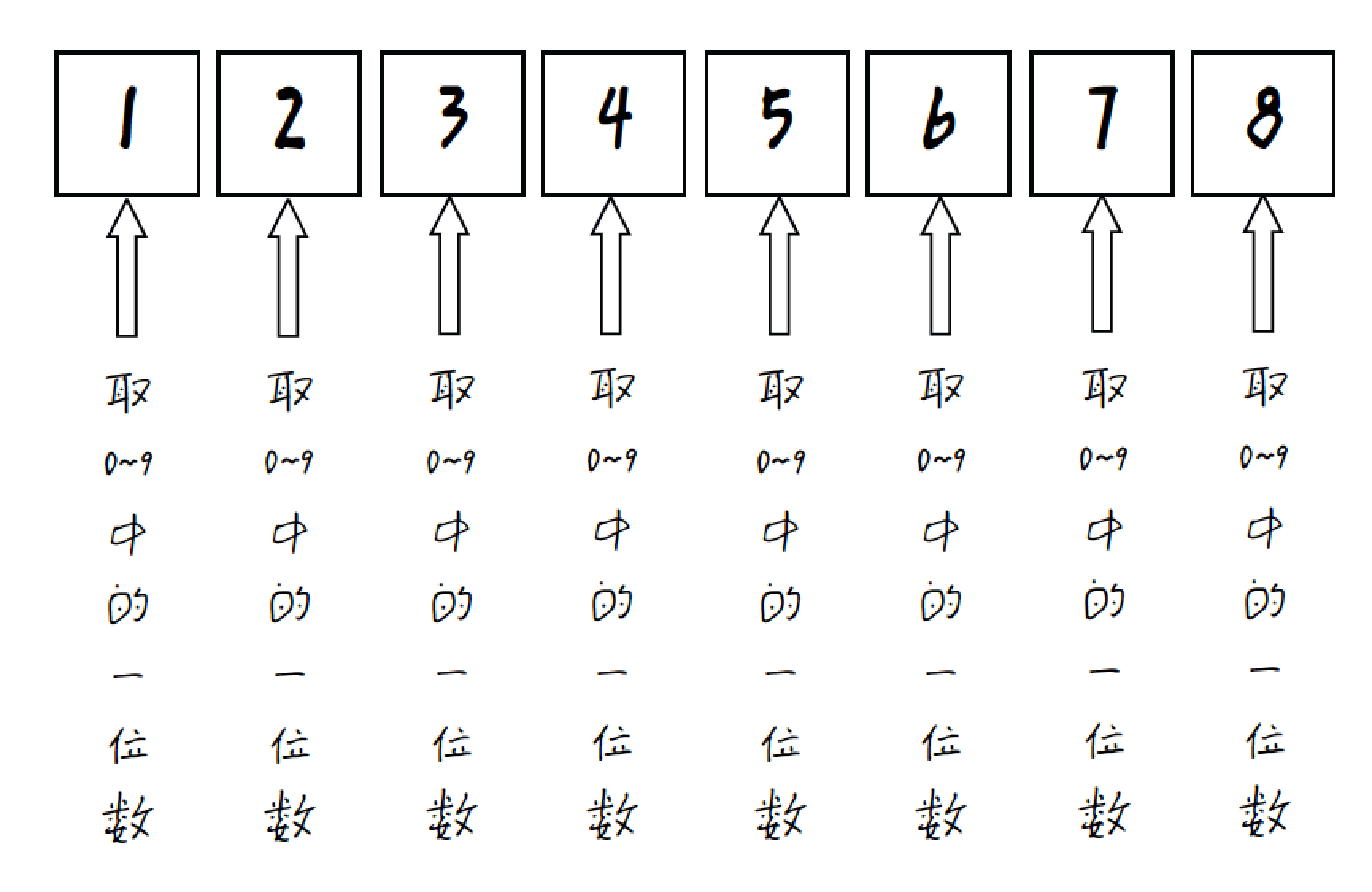
的顺序排成一列，叫做 n 个不同元素的一个 m-可重排列。

1. 有多少个电话号码？

对于重复排列中排列数的计算，不能再使用前面介绍的方法了。

例如：北京市固定电话号码是由 8 位数字组成，每位数字可从 0～9 中任取一个，问北京市固定电话最多可有多少种不同的电话号码？

不考虑电话号码中是否有特殊号码，只按数字进行排列的话，8 位数字中每一位数都可以在 0～9 这 10 个数码中取一个。



可按以下方法计算电话号码各位的可能情况：

第 1 位：可取 0~9 这 10 个数中的一个，有 10 种可能；

第 2 位：可取 0~9 这 10 个数中的一个，有 10 种可能；

第 3 位：可取 0~9 这 10 个数中的一个，有 10 种可能；

…… ……

8 位电话号码可分 8 步分别考虑可能的情况，根据乘法原理可得：

$$=10\times10\times10\times10\times10\times10\times10\times10$$

$$=100000000=10^8$$

因此，8 位数的电话号码有 1 亿种可能。也就是说，用 8 位数字可以排列出 1 亿个电话号码。

注意，在上面的排列中，将各种情况都包含在内，没有考虑一些特殊情况。例如，8 位数全部为 0，或前 7 位都为 0，只有个位数的（即 1 位数的电话号码），或者只有 2 位数电话号码，这些在实际生活中都不存在。

从这个例子可看出，对于可重排列，要计算其排列数，可使用以下公式：

$$n^m$$

即，n 个不同元素的 m-可重排列数为 n 的 m 次。

2．汽车牌照问题

在国内，民用机动车的牌照号是由多部分组成的，有省的简称，地级市字母代码及 5 位字符编码组成，如图 4-17 所示的车牌号为“浙 F99999”。



图 4-17

如果车牌号后面的 5 位数只采用数字进行组合，可以有多少个不同的号牌？

根据上面可重排列的计算规则，可得：

$$10^5=100000$$

即，可有 10 万个不同的号牌。

对于一个地级市，10 万个车牌号明显不够用。为了使每一位车主都能有自己唯一的车牌号，该怎么办呢？

根据可重排列数的计算公式中用到的 n 和 m 这两个变量，可有两种解决方案：

第一种方法是增加 m 的值（即增加车牌号码的位数，由 5 位数改为 6 位、7 位、甚至 8 位数），这样就可呈 10 倍、100 倍、1000 倍地增加不同的车牌号。

第二种方法则是增加 n 的值，也同样可增加不同的车牌号。

现实中采用是的第二种方法，在车牌号后 5 位数中除了使用数字之外，增加使用英文字母。除了英文字母 O 与数字 0，字母 I 与数字 1 不易分辨不用之外，使用其余 24 个大写字母。这样，10 个数字加上 24 个字母，共 34 个符号，则可得到 4 千多万个不同的号牌。

$$34^5=45435424$$

可以看到，增加 n 的值，可显著增多可重排列数。

现在一般一个地级市的机动车拥有量在百万之内（整个北京 400 万辆左右，分到各区就只有几十万辆了），肯定够用了。一般城市通常最多在车牌号码的 5 位中使用 2 位字母，那么，这样会有多少个不同的车牌号码呢？下面分 3 种情况来计算：

第 1 种，5 位号码全部为 0～9 的数字，可有 10 万个不同的车牌号。

第 2 种，若在 5 位车牌号码中有一位使用英文字母，其他 4 位使用数字，可有以下 5 类情况（字母 A 表示 24 个英文字母中的一个，数字 0 表示 0～9 中的一个数字）。

A0000、 0A000、 00A00、 000A0、 0000A

每类可得到不同车牌号码数量为：

$$24 \times 10 \times 10 \times 10 \times 10=240000$$

则 5 类情况可得到不同车牌号码数量为：

$$240000 \times 5=1200000$$

也就是说，第二种车牌号码（即 1 位英文字母、4 位数字）可有 120 万个不同的号码。

第 3 种，若在 5 位车牌号码中有 2 位使用英文字母，其他 3 位使用数字，可有以下 10 类情况（字母 A 表示 24 个英文字母中的一个，数字 0 表示 0～9 中的一个数字）：

AA000、　A0A00、　A00A0、　A000A

0AA00、　0A0A0、　0A00A

00AA0、　00A0A

000AA

每类可得到不同车牌号码数量为：

$$24 \times 24 \times 10 \times 10 \times 10=576000$$

则 10 类情况可得到不同车牌号码数量为：

$$576000 \times 10=5760000$$

将以上 3 种情况中的数值进行累加，即可得到不同车牌号码数量为：

$$100000+1200000+5760000=7960000$$

这样，5 位车牌号中，有 0～2 位用英文字母，另 5～3 位使用数字，可有 796 万多个不同的号码，足够使用了。

4.5　计算机中的字符编码

通过将数字、字符进行一定的排列，可得到如电话号码、车牌号这类日常用到的信息。类似地，在计算机中也广泛使用这类可重排列进行信息处理。例如，计算机中的字符编码就是一个明显的例子。在计算机中，最初使用的是 ASCII 码，但是，由于这种编码只能处理英文字符，对于像中文这种有上万种不同字符的情况，就无法处理了。为此，必须引用新的编码，常见的如 GB2312、UNICODE 等。

4.5.1　ASCII 码能表示的字符数量

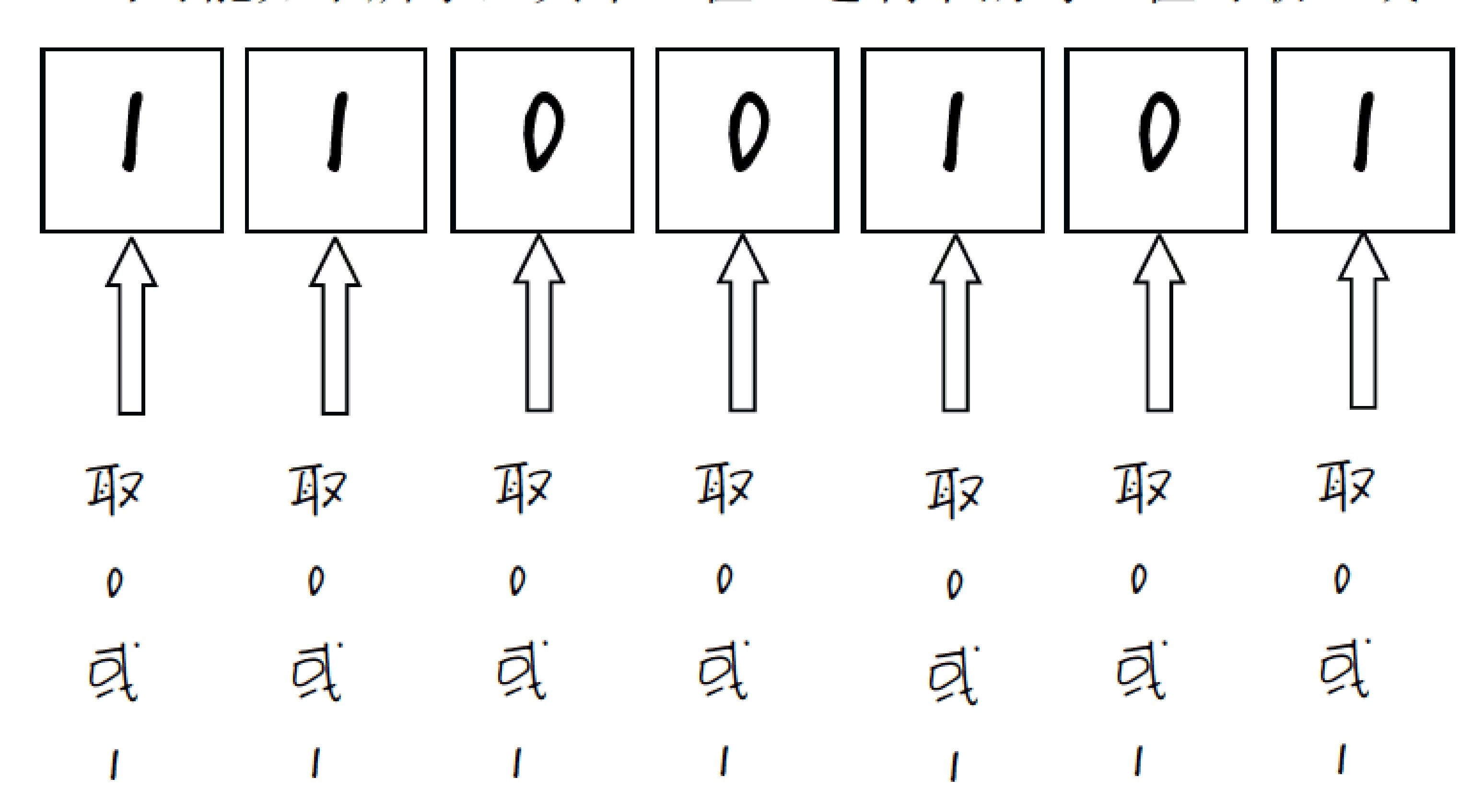
信息在计算机上是用二进制表示的，这种表示法让人理解起来很困难。因此计算机上都配有输入和输出设备，这些设备的主要目的就是，以一种人类可阅读的形式将信息在这些设备上显示出来，供人们阅读理解。为保证人类和设备、设备和设备之间能进行正确的信息交换，人们编制了统一的信息交换代码，这就是 ASCII 码，它的全称是“American Standard Code for Informatio”（美国信息交换标准代码）。

1．标准ASCII码

ASCII 是基于拉丁字母的一套电脑编码系统。它主要用于显示现代英语和其他西欧语言，是现今最通用的单字节编码系统，并等同于国际标准 ISO/IEC 646。

在 ASCII 码的最初编码规则中，规定使用 7 位二进制来表示一个字符。那么，按这种规则，ASCII 码有多少个不同的编码呢？也就是说能表示多少个不同的信息呢？

一个 ASCII 码可能如下所示，其中 8 位二进制中的每一位可取 0 或 1。



根据前面学习的计算可重排列数的规则来计算，每一位二进制位有 2 种可能（即 n=2），其位数为 7（即 m=7），则：

$$n^m=2^7=128$$

即，ASCII 码可表示 128 种不同的信息。如表 4-1 所示就是计算机中用到的 ASCII 码表，每一个编码对应一个控制字符或一个英文字母、数字、特殊符号。

从表 4-1 的 ASCII 码表中可发现一些编码的大小规则：

- 数字 0～9 比字母的编码小，如"9"<"A"；
- 数字 0 比数字 9 的编码小，并按 0～9 顺序递增，如"2"<"3"；
- 字母 A 比字母 Z 的编码小，并按 A 到 Z 顺序递增，如"A"<"B"；
- 相同字母的大写字母比小写字母编码小，如"A"<"a"。

无论在哪种程序设计语言中，ASCII 的编码规则都是相同的。在程序中比较两个字符串的大小时，默认都是按各字符对应的 ASCII 码的大小来进行比较。因此，程序员应

记住以上规则。

根据以上编码的大小规则，只要记住数字“0”和字母“A”、“a”的 ASCII 码分别为 48、65、97，就可以推导出后续数字或字母的 ASCII 码了。

表 4-1　ASCII码表

ASCII 码	控制字符	ASCII 码	字符	ASCII 码	字符	ASCII 码	字符
000	NUL	032	(space)	064	@	096	`
001	SOH	033	!	065	A	097	a
002	STX	034	"	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	$	068	D	100	d
005	END	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	’	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	＋	075	K	107	k
012	FF	044	,	076	L	108	l
013	CR	045	—	077	M	109	m
014	SO	046	。	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	：	090	Z	122	z
027	ESC	059	；	091	[	123	{
028	FS	060	<	092	\	124	|
029	GS	061	=	093	]	125	}
030	RS	062	>	094	^	126	~
031	US	063	?	095	_	127	□

2. 扩展ASCII码

ASCII 是美国标准，所以它不能很好地满足其他非英语国家的需要。例如英国的英镑符号（￡）在表 4-1 中就没有。

在使用过程中还发现 ASCII 码有以下问题：

- ❑ 没有拉丁语字母表重音符号。
- ❑ 没有斯拉夫字母表的希腊语、希伯来语、阿拉伯语和俄语中的符号。

- ❑ 没有中国汉字系统的象形汉字符号。
- ❑ 没有日本和朝鲜文符号。

为了适应计算机的发展，逐渐出现了扩展 ASCII 码。由于计算机中一个字节为 8 位二进制，而标准 ASCII 码只使用了 7 位。将 8 位二进制都用来表示 ASCII 码，就可扩展其表示范围。8 位二进制可表示的编码数量为：

$$n^M=2^8=256$$

增加一位二进制，可扩充一倍的编码数量。

扩展 ASCII 码表除了前面 0～127 这部分与表 4-1 的标准 ASCII 码相同之外，还增加了 128～255 这 128 个编码，如表 4-2 所示。

表 4-2　扩展ASCII码表

ASCII 码	字符	ASCII 码	字符	ASCII 码	字符	ASCII 码	字符
128	Ç	160	á	192	└	224	α
129	ü	161	í	193	┴	225	β
130	é	162	ó	194	┬	226	Γ
131	â	163	ú	195	├	227	π
132	ä	164	ñ	196	─	228	Σ
133	à	165	Ñ	197	┼	229	σ
134	å	166	ª	198	╞	230	μ
135	ç	167	º	199	╟	231	τ
136	ê	168	¿	200	╚	232	Φ
137	ë	169	⌐	201	╔	233	Θ
138	è	170	¬	202	╩	234	Ω
139	ï	171	½	203	╦	235	δ
140	î	172	¼	204	╠	236	ω
141	ì	173	¡	205	═	237	ϕ
142	Ä	174	«	206	╬	238	ε
143	Å	175	»	207	╧	239	⌒
144	É	176	░	208	╨	240	≡
145	æ	177	▒	209	╤	241	±
146	Æ	178	▓	210	╥	242	≥
147	ô	179	│	211	╙	243	≤
148	ö	180	┤	212	╘	244	∫
149	ò	181	╡	213	╒	245	ι
150	û	182	╢	214	╓	246	÷
151	ù	183	╖	215	╫	247	≈
152	ÿ	184	╕	216	╪	248	o
153	Ö	185	╣	217	┘	249	•
154	Ü	186	║	218	┌	250	·
155	¢	187	╗	219	█	251	√
156	£	188	╝	220	▄	252	∩
157	¥	189	╜	221	▌	253	²
158	¶	190	╛	222	▐	254	■
159	ƒ	191	┐	223	▀	255	

4.5.2　能表示更大范围的编码

参考表 4-1 和表 4-2 可看出，包含扩展 ASCII 码在内，一共只能表示 256 个符号，只包括了一些字符、数字、标点符号的信息表示。这主要是因为计算机是美国发明的，在英文下面，我们使用 ASCII 码表示就足够了！但是在汉字输入时，用 ASCII 码就不能表示所有的汉字了（仅常用汉字就上万个）。

1．用双字节和4字节进行编码

怎么来唯一地表示成千上万的汉字呢？根据我们前面介绍的排列规则，由于计算机中信息只能使用二进制表示，每一个二进制位只能为 0 或 1（相当于在排列中的 n 值为固定的 2），因此，只有增加二进制位数（相当于增加 m 的值），这样才能扩大排列数，以表示更多的汉字或其他符号。

在现在的计算机中，虽然最小处理单位为一个二进制位，但在读取或存储数据时，通常都是按字节进行处理的。一个 ASCII 码正好是一个字节（8 位二进制），要扩展字符的编码，就可考虑采用 2 个字节（16 位二进制位），则可表示的符号数为：

$$n^M=2^{16}=65535$$

一般来说，65536 个排列数应该能够保存常用的汉字和符号了。所以，现在常见的中文编码（有很多中文编码方案）都采用 2 个字节来表示一个汉字，称为双字节形式。

可是，双字节汉字编码仍然不够用。汉字的数量非常庞大，总量是多少，大家没有一个统一的说法。有“总汇汉字之大成”评价的《康熙字典》，收录的汉字是 4 万多个。1994 年出版的《中华字海》收入了 87019 个汉字，而已经通过专家鉴定的北京国安咨询设备公司的汉字字库，收录有出处的汉字 91251 个，据称是目前全国最全的字库。

汉字数量	
康熙字典	>40000
中华字海	87019
某公司汉字字库	91251

可以看出，双字节编码仍然不够用，因此又出现了 4 字节编码。4 个字节共 32 位二进制位，可排列出的数量为：

$$n^M=2^{32}=4294967296$$

4 字节编码就足够大了，排列数可达到 40 多亿，可以将世界上所有语言的符号都进

行唯一编码。

2. 认识Unicode

Unicode（统一码、万国码、单一码）是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。Unicode 于 1990 年开始研发，1994 年正式公布。随着计算机工作能力的增强，Unicode 也在面世以来得到迅速普及。

Unicode 是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode 用数字 0~0x10FFFF 来映射这些字符，最多可以容纳 1114112 个字符，或者说有 1114112 个码位。码位就是可以分配给字符的数字。

在 Word 中打开“符号”对话框，可看到汉字或符号的 Unicode 编码，如图 4-18 所示。

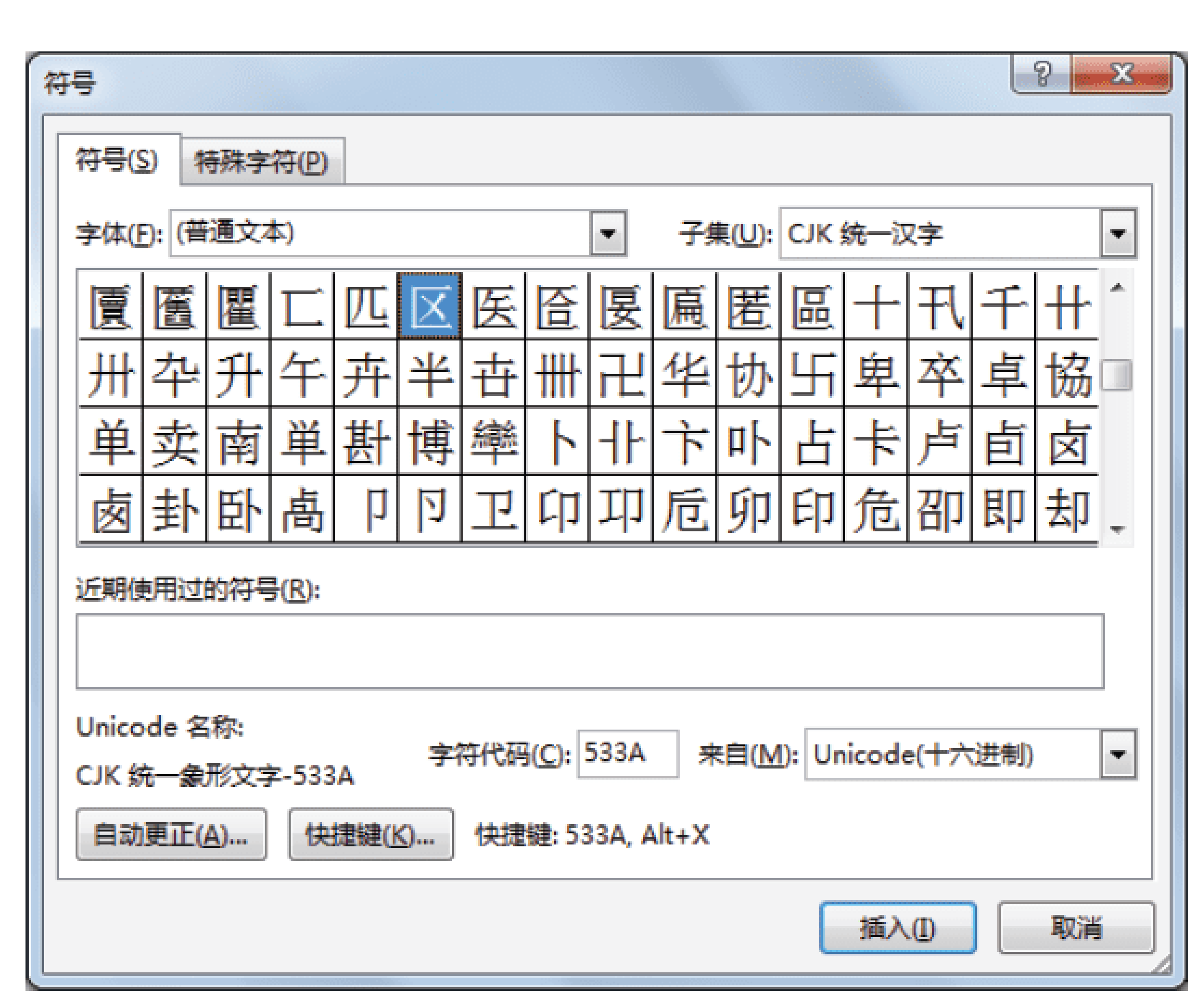


图 4-18

其实在 Unicode 之前，一共存在过 3 套中文编码标准，分别是：

- GB2312-80：是中国大陆使用的国家标准，其中一共编码了 6763 个常用简体汉字。
- Big5：是台湾使用的编码标准，编码了台湾使用的繁体汉字，大概有 8 千多个。
- HKSCS：是中国香港使用的编码标准，字体也是繁体，但跟 Big5 有所不同。

这些编码标准为了表示上万的汉字，都是使用了 16 位二进制位的排列来进行编码。虽然这 3 套编码标准都采用了两个扩展 ASCII 的方法来表示一个汉字，但它们编码区间各不相同，因此这 3 套编码互不兼容。这就导致在同一个系统中同时显示 GB 和 Big5 基本上是不可能的。

4.6　密码的长度

对于可重复排列，在日常生活应用中一个有趣的话题就是密码长度的问题：什么样的密码是安全的？增加密码的长度？还是增加密码可用字符数？

4.6.1　容易破解的密码

在日常生活中，我们的银行账号要设置密码、邮箱要设置密码、使用手机客服需要密码等。总之，现在是一个充满密码的时代。

首先，我们来看容易破解的密码。

想象一下，我们打开保险柜的密码如果设置为这种情况：一个单刀双掷开关，如图 4-19 所示。有两种状态，向左边接通或向右边接通。

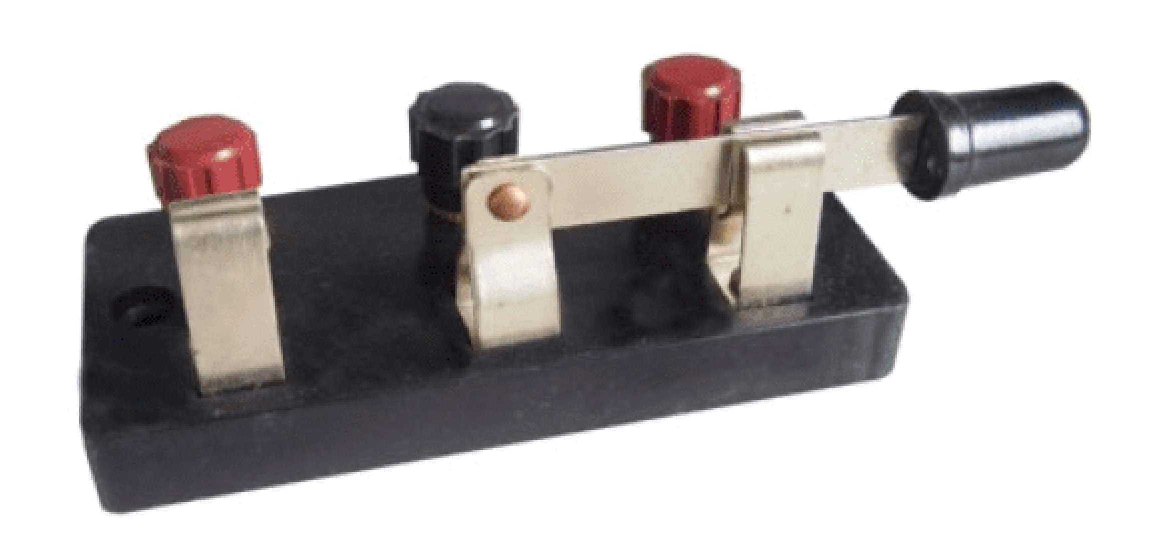


图 4-19

这种保险柜能保险吗？我们只需要向左接通试一下，若不能打开，再向右接通。只有两种状态，并且只有一个开关，则其排列数为 2。因此，最多只需要试两次就可进行破解。

现实中当然没有这么简单的密码情况。例如，我们到银行取款时需要输入密码，现在银行柜台和 ATM 柜员机中使用的密码只能为数字。如果密码只设置为 1 位数字，则 0～9 这 10 个数字用 1 位数可得的排列数为：

$$n^M=10^1=10$$

即，如果密码设置为 1 位数字，只有 10 种可能。显然，这种密码就很不安全，很容易被破解。最多连续重试 10 次就可枚举完所有密码。

为了增强密码的安全性，可以从两个角度来考虑：一是增加密码的长度，这样，就使可用密码的数量呈几何级数增长，增加被其他人通过枚举方法进行破解的难度。第二种方法是增加可选择的字符，同样可增加可用密码的数量。

目前，我国银行使用的 ATM 只能输入数字，因此增加可选择字符这个方案行不通，就只有通过增加密码长度来保证密码的安全了。现在银行密码的最大长度为 6，因此，

可使用密码的数量为：

$$n^M=10^6=1000000$$

即，如果破解者在最坏的情况下，可能需要枚举 100 万次，才能知道某银行账户的密码。由于在 ATM 柜员机上只能手工输入密码，且一天之内如果密码输入错误 3 次，该账号将被锁住，这样就增加了银行账号的安全性。因此，6 位的数字密码具有一定的安全性。

当然，破解者通常不会仅通过枚举方法来枚举所有可能的密码，一般还会根据账号所有者的个人信息进行破解。所以，我们在设置银行账号密码时应尽量不用自己的个人信息（如生日、身份证号中的一部分等），以防被有心人快速破解。

4.6.2 多长的密码才安全

由前面的分析可看出，银行 ATM 的密码为 6 位数字，有 100 万种可能。由于银行采用了一些限制条件，可保证账号的安全性。

可是，如果这种 6 位数字密码用在计算机中，就显得不足了，太容易被别人破解了。

例如，对于登录邮箱时输入的密码，如果只设置为 6 位数字，则破解者可通过计算机的快速运行能力，编写程序，在很短的时间内就可将 100 万种密码逐一测试，找出正确的密码。这样，邮箱账号的密码就没有安全性可言了。

根据前面的分析，可以增加密码的长度，以增强密码的安全性。那么，多长的密码才安全呢？

密码被破解的可能性不仅与密码的长度相关，与计算机的发展也相关。对于相同长度的密码，计算机的速度越快，被破解的可能性就越大。

例如，很多论坛注册账号时都要求密码的长度要大于 8，与 6 位数的密码相比（如果只是使用数字 0～9 作为密码），可用密码数量会增加 100 倍：

$$n^M=10^6=1000000$$

$$n^M=10^8=100000000$$

因此，我们在计算机中设置密码时，应尽量使密码设置得更长，以增加密码被破解的难度。

4.6.3 密码中使用的字符数量也很关键

要增加密码被破解的难度，就需要增加可用密码的数量。

前面说的增加密码的长度，只是增加计算可重复排列数中的指数 m 的值，也可考虑增加基数 n 的值，这样也可增加可用密码数量。

从表 4-1 中可看到，从 ASCII 码值为 32 开始至 126，一共有 95 个可由键盘直接输入的字符，若这些字符都可用来设置密码，则 6 位长度密码的可重复排列数可达到：

$$n^M=95^6=735091890625$$

破解者若想枚举完这 7350 多亿个密码，按目前计算机的速度，每秒钟大概能测试验证几百个密码，则要想将这 7350 多亿个密码枚举完成，24 小时开机都需要 20 多年的时间才能完成。

如果将密码长度设置得更长，如设置为 8 位，则可重复排列数可达到：

$$n^M=95^8=6634204312890625$$

这是一个天文数字，与使用 6 位长度的密码相比，可重复排列数增加了 9000 多倍，同样，其破解难度也就增加了 9000 多倍。

当然，以上我们计算的都只是理论上的值。一般情况下，系统为了使密码达到一定的安全性，都要求密码中必须包含数字、字母和特殊符号，并要求密码必须达到一定的长度。可是，很多用户为了方便自己记忆，在设置密码时可能只使用数字，或只使用字母，并且设置的长度往往也只达到系统要求的最底长度。这样，就导致自己的密码不安全。

因此，作为程序员，我们在系统中应对用户设置的密码进行检查，包括检查密码的长度、检查密码中包含的字符种类（至少应该要求用户使用两类字符）。作为用户，我们在设置自己账号的密码时，无论系统是否对密码的字符、长度有要求，我们都应设置足够的密码长度，并尽量使用数字、字母和特殊符号的组合来设置密码。这样，可使自己的密码更具有安全性。

第 5 章　余数——数据分组

在算术运算中，当两个整数相除的结果不能以整数商表示时，余数便是其“余留下的量”。当余数为 0 时，被称为整除。在数学中余数具有重要的作用，同样，在程序设计中，余数的作用也非常重要，如奇偶校验就是利用余数规则进行的。

5.1　复习小学的余数

我们在小学就开始学习余数，下面先来回忆一下小学时学的余数，然后再研究余数的性质，了解如何利用余数分组。

5.1.1　自然数的余数

有 15 盆鲜花要布置到会场，每行摆 5 盆，可以摆几行？如图 5-1 所示。



图 5-1

小学的计算题，当然很容易计算了：

$$15 \div 5=3$$

如果有 23 盆花，又能摆几行呢？

$$23 \div 5=4(\text{行}) \cdots 3(\text{盆})$$

根据上面的计算结果可看出，23 盆花摆 5 行多出来 3 盆，如果摆 5 行又少 2 盆。

以上算式中，计算出来摆 4 行后多出来的 3 盆，就是余数。

下面对余数做一个正式的定义：如果 a 和 d 是两个自然数，d 不等于 0，可以证明存在两个唯一的整数 q 和 r，满足 $a = qd + r$，且 $0 \leqslant r < d$。其中，q 被称为商，r 被称为余数。

当被除数小于除数时，我们以被除数为余数。例如：

$$7 \div 8=0 \cdots 7$$

表示 7 除以 8 的余数为 7。

另外需要注意，按上面的定义，可能导致两种可能的余数。例如，有以下除法式子：

$$-32 \div (-6)=5 \times (-6)-2$$

也可写成以下形式：

$$-32 \div (-6)=6 \times (-6)+4$$

这样，–32 除以–6 就有 2 个余数，分别为–2 和 4。

可是，这种对余数不明确的定义可能导致严重的计算问题，对于处理关键任务的系统，错误的选择会导致严重的后果。在一些系统中，会有特殊的除法指令，设定余数和被除数必须同号，这样就解决了多个余数的情况。

在上面的例子中，负余数（–2）为正余数（4）减 6 得到，6 即是除数 d，通常，当除以 d 时，如果正余数为 r_1，负余数为 r_2，那么：

$$r_1=r_2+d$$

5.1.2　余数的性质

在介绍用余数进行分组前，先了解余数的性质。

1．性质1

余数的性质 1：余数小于除数。

这是很明显的，如果余数可以大于除数，则余数就不具有唯一性了。例如，下面的除法算式中，余数出现多种情况。按性质 1 的要求，则只有最后一个算式是正确的，因为这个算式中余数为 3，小于除数 5，而其他算式中余数都大于除数 5。

$$23 \div 5=0 \cdots 23$$

$$23 \div 5=1 \cdots 18$$

$$23 \div 5=2 \cdots 13$$

$$23 \div 5=3 \cdots 8$$

$$23 \div 5=4 \cdots 3$$

2. 性质2

余数的性质 2：被除数、除数、商和余数之间的关系如以下所列公式，其中 a 为被除数，d 为除数，q 为商，r 为余数：

$$a=d \times q+r$$

$$d=(a-r) \div q$$

$$q=(a-r) \div d$$

$$r=q-d \times q$$

3. 性质3

余数的性质 3：有自然数 a、b、c，如果 a、b 除以 c 的余数相同，那么 a 与 b 的差能被 c 整除。即，如果有：

$$a \div c=x \cdots d$$

$$b \div c=y \cdots d$$

则有：

$$(a-b) \div c=z \cdots 0$$

例如：17、11 除以 3 的余数都为 2：

$$17 \div 3=5 \cdots 2$$

$$11 \div 3=3 \cdots 2$$

则，（17-11）的差能被 3 整除：

$$(17-11) \div 3=2 \cdots 0$$

4．性质4

余数的性质 4：a 与 b 的和（a+b）除以 c 得到的余数（a、b 两数除以 c 在没有余数的情况下除外），等于 a、b 分别除以 c 得到的余数之和（或这个和除以 c 的余数）。

即，如果：

$$(a+b)\div c=z\cdots d$$

则有：

$$a\div c=x\cdots d_1$$

$$b\div c=y\cdots d_2$$

$$d=d_1+d_2$$

例如：整数 33、26 除以 5：

$$33\div 5=6\cdots 3$$

$$26\div 5=5\cdots 1$$

则：

$$(33+26)\div 5=11\cdots 4$$

其中的余数 4，是前面两式余数 3+1 之和。

当余数之和大于除数时，所求余数等于余数之和再除以 c 的余数。

例如：

$$33\div 5=6\cdots 3$$

$$24\div 5=4\cdots 4$$

以上两个算式的余数相加等于 7，这个余数之和大于除数 5，因此，(33+24）之后除以 5 的余数应等于（3+4）除以 5 的余数。

$$(33+24)\div 5=11\cdots 2$$

$$7\div 5=1\cdots 2$$

5．性质5

余数的性质 5：a 与 b 的乘积除以 c 所得的余数，等于 a，b 分别除以 c 所得的余数

之积（或这个积除以 c 的余数）。

即，如果：

$$(a \times b) \div c = z \cdots d$$

则有：

$$a \div c = x \cdots d_1$$

$$b \div c = y \cdots d_2$$

$$d = d_1 \times d_2$$

例如：整数 33、26 除以 5 的余数分别为 3 和 1：

$$33 \div 5 = 6 \cdots 3$$

$$26 \div 5 = 5 \cdots 1$$

因此：

$$(33 \times 26) \div 5 = 171 \cdots 3$$

可以看出，其余数等于 3×1（即 33、26 两数分别除以 5 所得的余数之积）。

当余数之积大于除数时，所求余数等于余数之积再除以 c 所得的余数。

例如：33、24 除以 5 的余数分别是 3 和 4，所以：

$$(33 \times 24) \div 5 = 158 \cdots 2$$

$$(3 \times 4) \div 5 = 2 \cdots 2$$

可以看出，（33×24）除以 5 的余数等于（3×4）除以 5 的余数。

5.1.3　用余数进行分组

用余数进行分组，是怎么分组的呢？

其实，除法操作就是一种分组操作。例如，若将 1～10 这 10 个数按奇偶性质分组，该怎么操作呢？

这时，只需要用 1～10 这 10 个数中的每一个数去除以 2，再根据余数为 0 或 1，分别归入不同的分组，若余数为 0，则为偶数，余数为 1，则为奇数。

数据	除以2的余数	偶数组	奇数组
1	1		√
2	0	√	
3	1		√
4	0	√	
5	1		√
6	0	√	
7	1		√
8	0	√	
9	1		√
10	0	√	

可以看出，通过余数即可将 1～10 这 10 个数分为两组，奇数组的数据分别为（1、3、5、7、9），偶数组的数据分别为（2、4、6、8、10）。

类似地，如果将一批数除以 5，则可通过得到的余数将这批数分为 5 组。也就是说，若要将一批数据分为 n 组，则只需要将该批数据中的数据逐个除以 n，然后根据余数即可完成分组。

5.2　日历中的数学

日历就是利用余数来进行分组的一个例子。一周有 7 天，因此可将日期分为 7 组，每组对应一周中的一天。这样，按组竖向排列，就可得到一份月历。

5.2.1　*n* 天后是星期几

如图 5-2 所示是 2013 年 5 月的月历，今天是 5 月 3 日，星期五，则 7 天之后（3+7）为 5 月 10 日也是星期五，而 14 天之后（3+7+7）为 5 月 17 日、21 天之后（3+7+7+7）为 5 月 24 日、28 天之后（3+7+7+7+7）为 5 月 31 日都是星期五。从图 5-2 中也可看到，这几天都位于月历星期五的同一列中。


2013年5月

日	一	二	三	四	五	六
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

图 5-2

知道了这个规则，则可以很容易地推算出 n 天之后是星期几。

例如，若想推算出 100 天之后是星期几，则可考虑首先到了接近第 100 天时为星期五的那一天：在今天（5 月 3 日）之后的第 7、14、21、28、35、42、49、56、63、70、77、84、91、98 天都为星期五。接着，可继续向下推算：

星期五：第 7、14、21、28、35、42、49、
56、63、70、77、84、91、98 天

星期六：第 99 天

星期日：第 100 天

由此可知，今天之后的第 100 天为星期日。

其实，不用按上面的推算过程，通过计算余数可以更方便地计算出 n 天后为星期几，计算公式如下：

$$n \div 7 = q \cdots r$$

在以上算式中，我们不关注商 q，只需要关注余数 r 即可。n 除以 7 后的余数在 0～6 这 7 种情况，因此可分为 7 组，将这 7 个余数排列起来，再与当前的星期数对应，即可快速推出 n 天后为星期几。

例如，在上面计算 100 天后为星期几，可按以下公式进行计算：

$$100 \div 7 = 14 \cdots 2$$

接着按下面的形式列出余数 0～6，将余数为 0 的下方填写与今日对应的星期数，然后向后循环，即可得到如图 5-3 所示余数与星期几的对应关系。

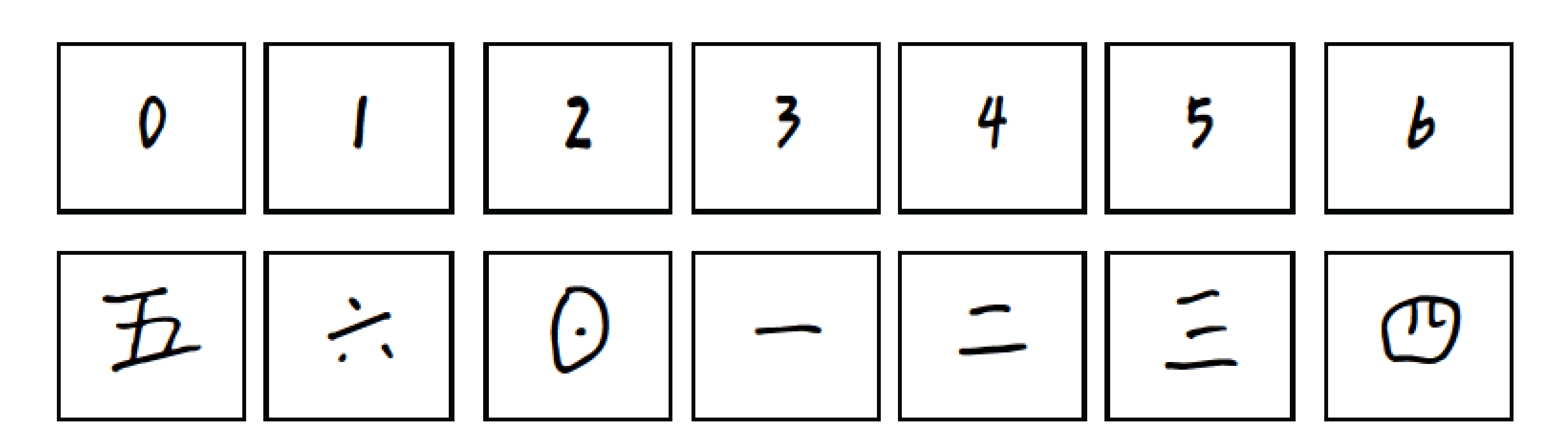


图 5-3

当余数为 2 时，从上面的对应关系中得到 100 天后为星期日。

有了这种通过余数来推算星期几的方式，则可计算出任意天数后为星期几。例如，今天为星期五，1000 天之后为星期几？

$$1000 \div 7 = 142 \cdots 6$$

根据图 5-3 所示对应关系可知，1000 天后为星期四。

需要注意的是，图 5-3 的对应关系是变化的，余数为 0 时对应的是当天的星期数。

5.2.2　下月的今天是星期几

如果要计算下月的今天是星期几，该怎么办？这时，由于没有距今天的天数，就不能直接使用前面介绍的方法进行求余数了。

其实，要计算星期几，最终还是必须转化到求 7 的余数上来。因此，要计算下月的“今天”是星期几，可以先计算下月的“今天”距离今天是多少天，然后就回到上面介绍的通过天数推算星期几的方法中了。

那么，下月的“今天”是怎么定义的呢？如果今天是 2013 年 5 月 3 日，下月的“今天”就是 2013 年 6 月 3 日，即只是增加月份，日期数值不变。不过，要考虑一个问题，就是如果今天是 2012 年 12 月 3 日，则下月的“今天”应该是 2013 年 1 月 3 日，年、月数值都要变化。

下月的“今天”距今天的天数是多少呢？

根据历法，每年的 1、3、5、7、8、10、12 月为大月，每月有 31 天，每年的 4、6、9、11 月为小月，每月 30 天，每年的 2 月的天数根据平年、闰年分别为 28、29 天。

28 天：平年 2 月
29 天：闰年 2 月
30 天：第 4、6、9、11 月
31 天：第 1、3、5、7、8、10、12 月

根据以上历法知识，若今天是 2013 年 5 月 3 日，则下月的“今天”为 6 月 3 日。由于 5 月有 31 天，因此今天距离下月“今天”的天数也为 31 天，则可通过以下算式计算出余数：

$$31 \div 7 = 4 \cdots 3$$

余数为 3，查询图 5-3 所示对照关系，可知道 2013 年 6 月 3 日为星期一。打开日历，可看到 2013 年 6 月的日历如图 5-4 所示，在这里可看到 2013 年 6 月 3 日为星期一。



图 5-4

5.2.3 10年后的“今天”是星期几

增加点难度，10年后的“今天”是星期几，该怎么计算？

理论上，同样只需要计算出10年后的今天距离“今天”的天数，就可以很简单地推算出是星期几。可是，跨度为1月的天数比较好计算，也就可以很容易地推算出下月的今天是星期几。如果要横跨几年，该怎么计算呢？

可以从另一个角度考虑，我们只要寻找一个求公元 n 年 m 月 k 日是星期几的一个公式即可。这样，就不必再去关注两个日期之间相距的天数了。

前面我们已经介绍过，根据月大、月小、平年、闰年等关系，每月的天数是不一样的。月大、月小是有规律可循的，平年、闰年有规律吗？

我们知道，公历的年是以寒暑更替的周期（俗称回归年，即太阳相继两次过春分点所经历的时间）为基础的，但是，一回归年有365天5时48分46秒，其天数不是整数。如果一年的天数不是整数，对人们来说，使用起来很不方便。为了解决这个矛盾，公历制定了相应的设置闰年的法则，根据该法则就可判断是否为闰年。设年份为 n，则判断 n 是否为闰年的法则如下：

- 如果 n 能被100整除，且能被400整除，则 n 是闰年；
- 如果 n 能被4整除，但不能被100整除，则 n 是闰年。

例如：

2000年是闰年（2000能被100和400整除）；

1996年是闰年（1996能被4整除，不能被100整除）；

1900年是平年（1900年能被100整除，但不能被400整除）。

根据历法知识我们知道，平年一年是365天，闰年一年是366天，则要求公元 n 年 m 月 k 日是星期几，只需要推算从公元1年至公元 n 年一共经历了多少个闰年。根据判断闰年的法则，可使用以下公式计算出 n 年共经历了多少年闰年：

$$\left[\frac{n-1}{4}\right]-\left[\frac{n-1}{100}\right]+\left[\frac{n-1}{400}\right]$$

在以上公式中，方括号表示计算的结果取整。以上算式中第一项表示能被4整除的年份数量，其中也包含了能被100整除的年份，但是能被4整除，且能被100整除的年份不是闰年，因此应减去能被100整除的年份（以上算式的第2项）；但是这时减去能被100整除的年份时，将能被400整除的年份数量也减掉了，而这部分年份是闰年，因此还要加回去，第3项就是得到能被400整数的年份数量。

由于是计算到 n 年经历了多少年闰年，当然就不能将 n 年包含在内，所以使用(n-1)，实际是判断从公元1年至公元 n-1年（含）共有多少个闰年。

根据上面的公式可推出下面的公式，计算出距离公元1年1月1日的天数。

$$S=365\times(n-1)+\left[\frac{n-1}{4}\right]-\left[\frac{n-1}{100}\right]+\left[\frac{n-1}{400}\right]+c$$

在上面公式中 *n* 为年份，c 为公元 n 年 m 月 k 日在 *n* 年中的天数。

通过以上公式计算出 S 值，再用 S 值除以 7，得到余数，余数为 1 则是星期一，余数为 2 则是星期二……，余数为 0 则为星期日，如图 5-5 所示（这是因为公元 1 年 1 月 1 日为星期一，所以推算得出图 5-5 所示的对应关系）。

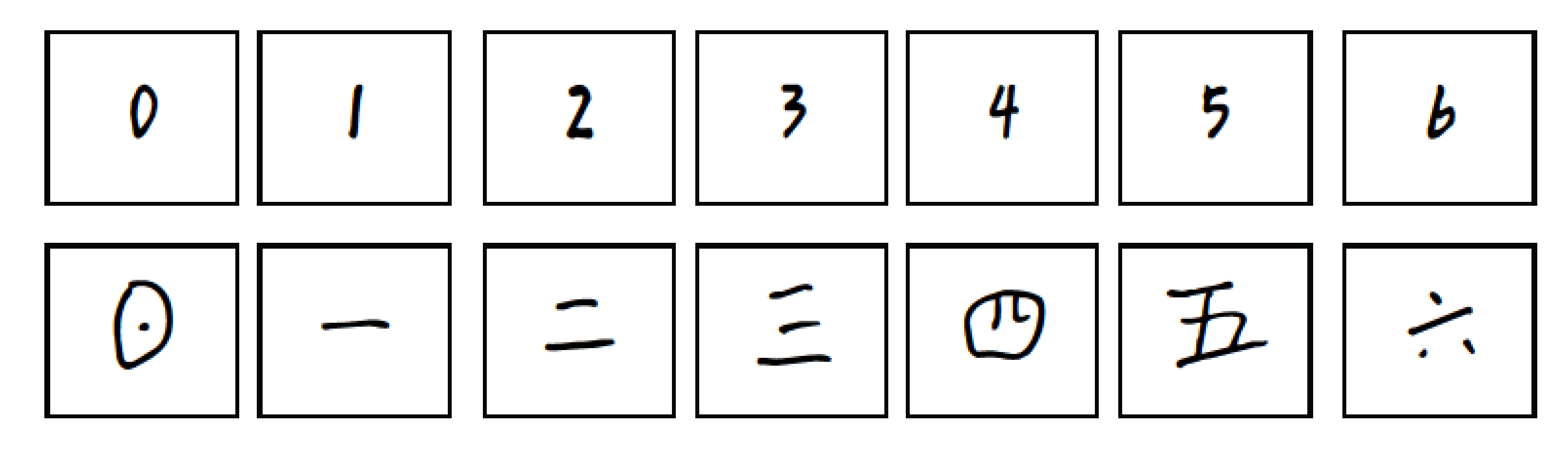


图 5-5

其实，上面的公式还可以简化为以下形式（即不乘 365）：

$$S=(n-1)+\left[\frac{n-1}{4}\right]-\left[\frac{n-1}{100}\right]+\left[\frac{n-1}{400}\right]+c$$

根据以上公式，计算 2013 年 1 月 1 日为星期几，首先计算 S 值：

$$S=(2013-1)+\left[\frac{2013-1}{4}\right]-\left[\frac{2013-1}{100}\right]+\left[\frac{2013-1}{400}\right]+1$$
$$=2501$$

接着将 S 的值除以 7，得到余数：

$$2501\div 7=357\cdots 2$$

所以，从图 5-5 中的对应关系可查得，2013 年 1 月 1 日为星期二。

有了这个公式，计算 10 年后的“今天”是星期几就简单了。假设今天为 2013 年 5 月 3 日，则 10 年后的“今天”就是 2023 年 5 月 3 日。将数据代入上面的公可计算出 S 的值：

$$S=(2013-1)+\left[\frac{2013-1}{4}\right]-\left[\frac{2013-1}{100}\right]+\left[\frac{2013-1}{400}\right]+2\times 31+28+30+3$$
$$=2635$$

在上面公式中，后面那一串算式是计算 2023 年 5 月 3 日的天数，其中，有 1、3 月为大月，每月 31 天，2 月为平月 28 天，4 月为小月 30 天，5 月还包含 3 天。

接着将 S 的值除以 7，取余数：

$$2635\div 7=376\cdots 3$$

从图 5-5 的对应关系可得出，10 年后的“今天”（2023-5-3）为星期三。查询日历可

看到，计算结果是正确的，如图 5-6 所示。



图 5-6

可见，不管怎么变化，求星期几的操作都是以求余数为基础的。

5.3　心灵感应魔术

余数在魔术表演中也经常应用，对很多物品排列的魔术，都可通过余数进行破解。本节将演示一个心灵感应魔术的破解过程。

5.3.1　一个小魔术

来看一个小魔术，这个魔术大家都能玩。在这个魔术中，表演者通过心灵感应，能知道观众心中选的是哪一张牌。

魔术表演的具体过程如下：

（1）魔术师拿出一叠扑克牌，将牌洗乱之后，在桌面上发 3 列牌。按照从左向右每列发一张的顺序发牌，得到如图 5-7 所示的 3 列牌。



图 5-7

（2）魔术师请一位观众自己在心里记住 3 列牌中任意一张牌，不要向魔术师说出记住的是哪一张（假设观众记住的是黑桃 A），只需要向魔术师说自己记的牌位于哪一列（黑桃 A 在第 1 列）。

（3）魔术师将 3 列牌收起来。收牌时，魔术师按照第 2、1、3 列的顺序收牌（即先将第 2 列的牌收回，再将第 1 列的牌收回放在下面，最后将第 3 列的牌收回放在下面），这时牌的排列顺序如图 5-8 所示。

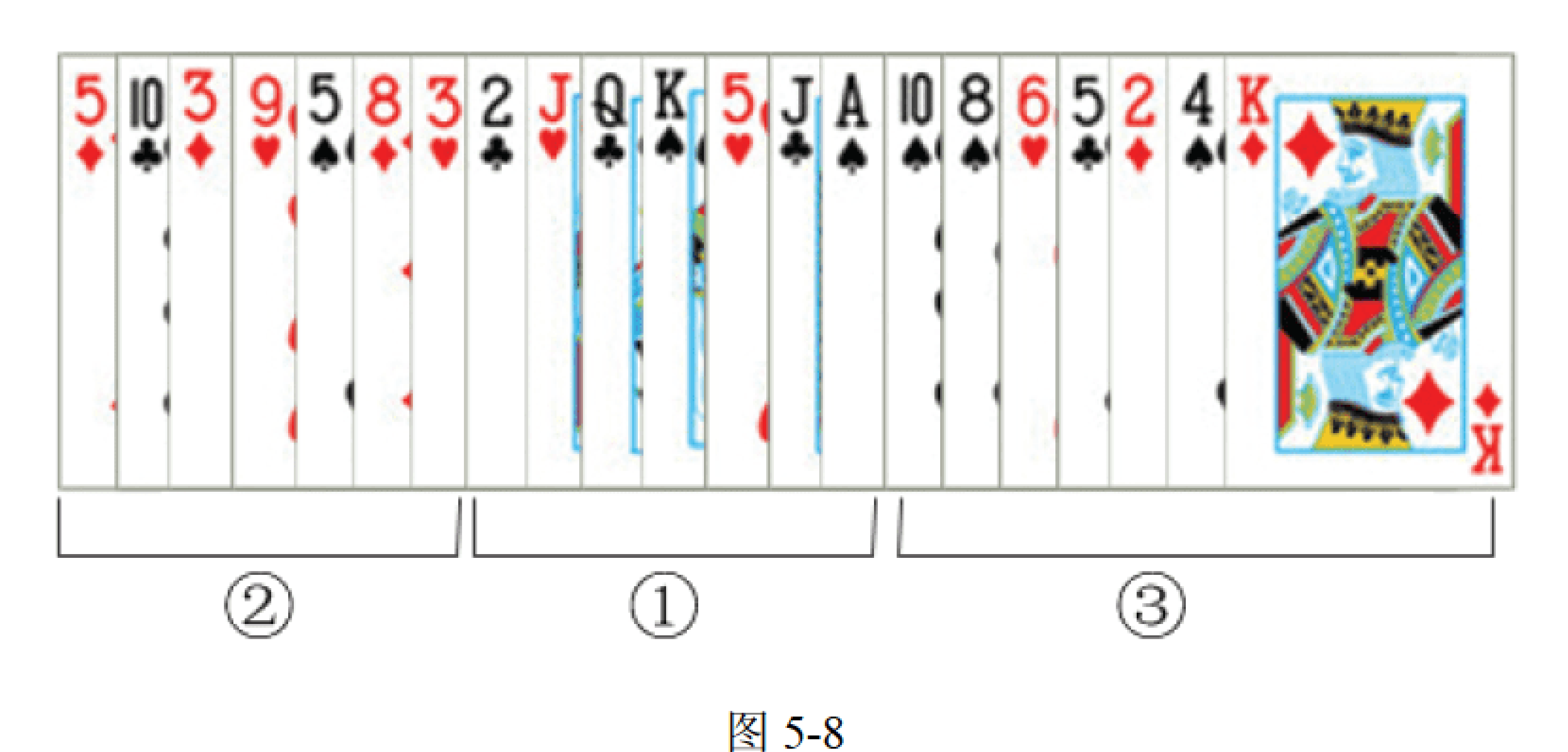


图 5-8

（4）魔术师又将这 21 张牌在桌面上发为 3 列，按照从左向右每列发一张的顺序发牌，得到如图 5-9 所示的 3 列牌，让观众说出自己记住的牌在哪一列（在图 5-9 中，黑桃 A 在第 2 列）。

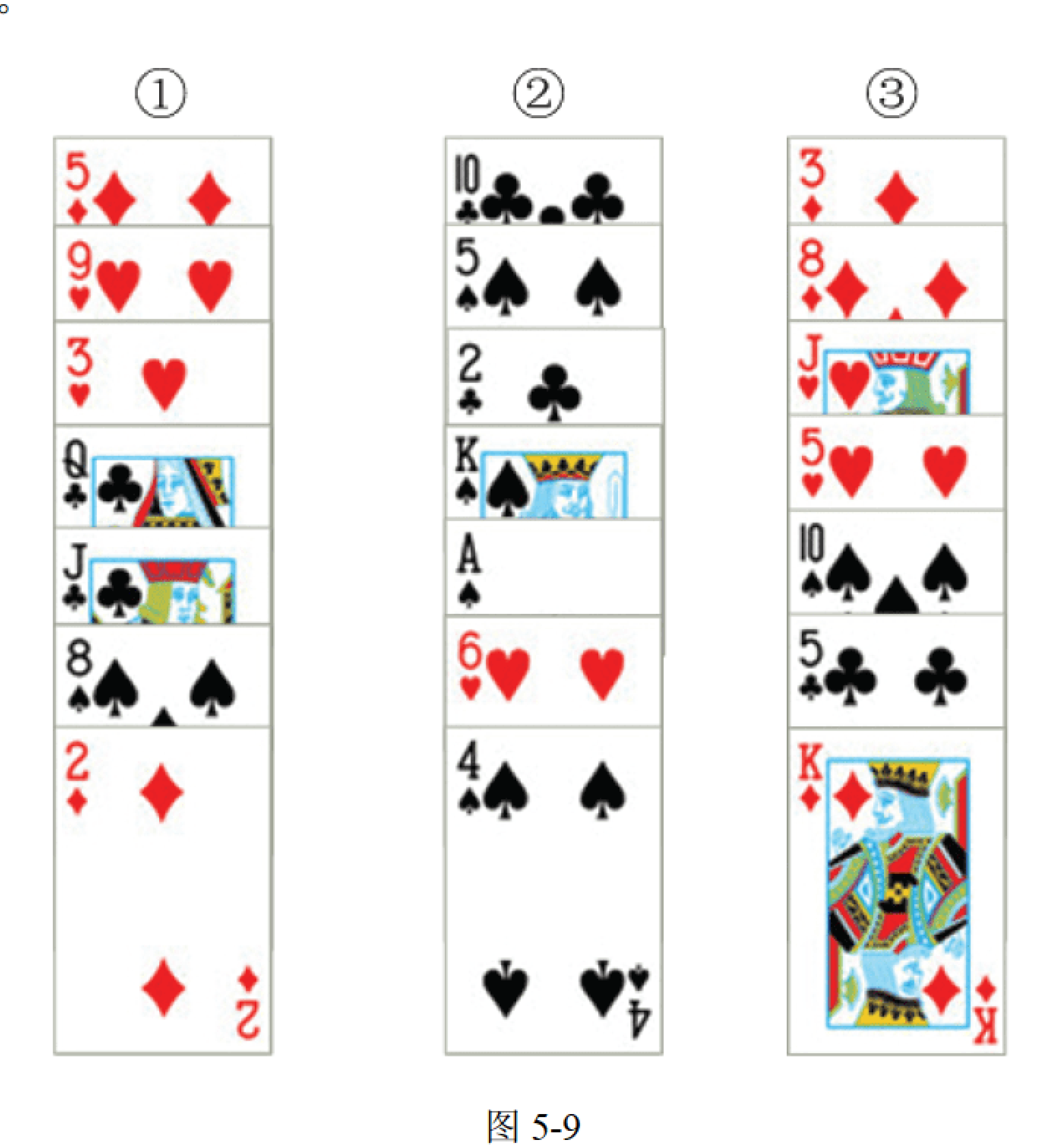


图 5-9

（5）魔术师按第 1、2、3 列的顺序将牌收集起来，这时，收集在一起的牌的排列顺

序如图 5-10 所示。

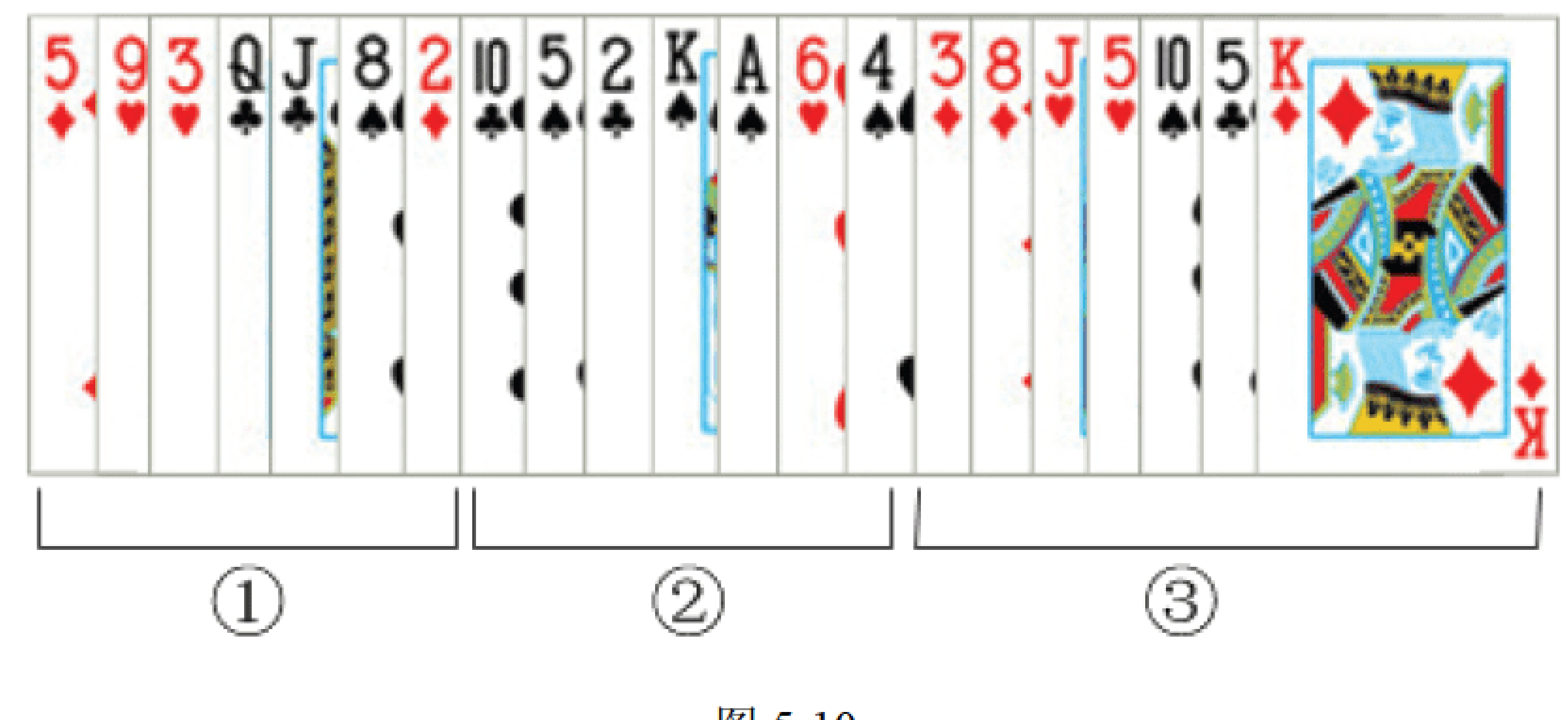


图 5-10

（6）魔术师又将这 21 张牌在桌面上发为 3 列，按照从左向右每列发一张的顺序发牌，得到如图 5-11 所示的 3 列牌。让观众说出自己记住的牌在哪一列（在图 5-11 中，黑桃 A 在第 3 列）。

（7）魔术师按照第 1、3、2 列的顺序将牌收集起来，这时，收集在一起的牌的排列顺序如图 5-12 所示。

（8）魔术师再次将这 21 张牌在桌面上发为 3 列，按照从左向右每列发一张的顺序发牌（这次是将牌面向下放置），然后，魔术师从第 2 列中抽出第 4 张牌，并向观众展示出这张牌，正是观众记住的黑桃 A。

将第（8）发的牌翻转过来看，3 列牌面如图 5-13 所示，可以看到，在图 5-13 中，第 2 列第 4 张正是黑桃 A。

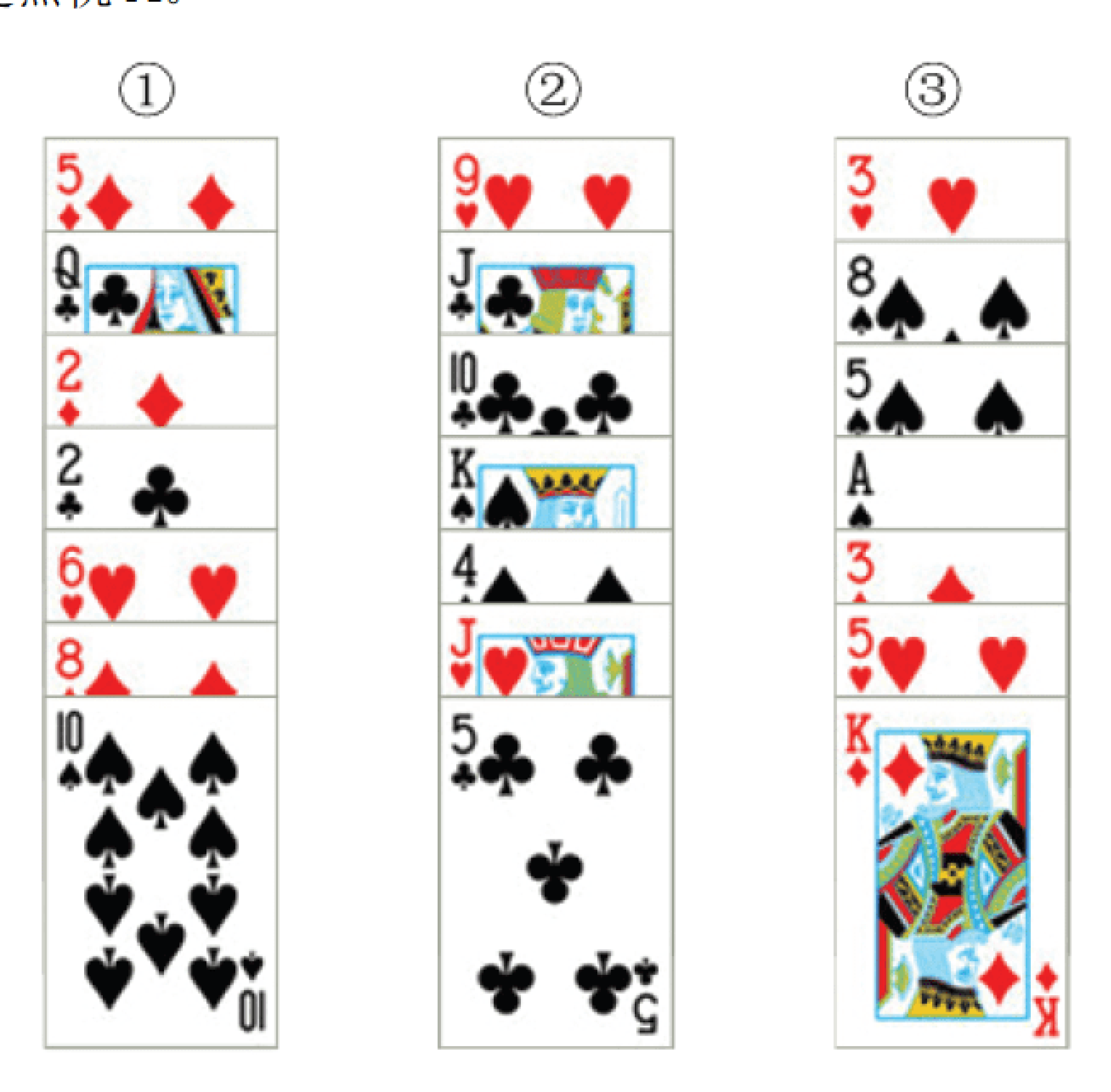


图 5-11

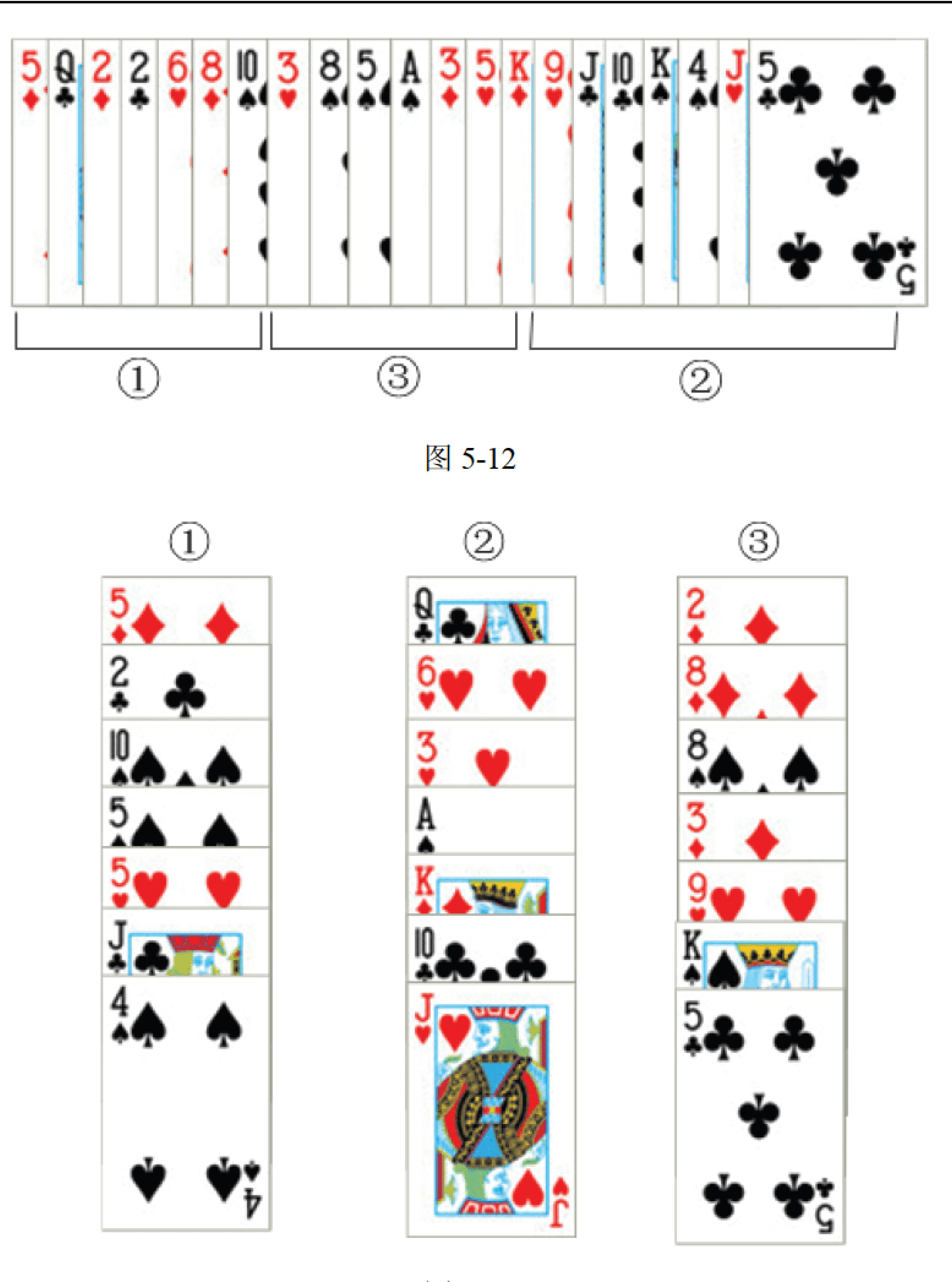


图 5-12

图 5-13

从以上魔术师的表演过程可看到，魔术师连着发了 4 次牌，让观众回答了 3 个问题，就将观众记住的牌移到了排列的正中（第 2 列第 4 张）。在表演时，很关键的一点就是将 3 列牌收集在一起时，要将观众指出的那一列的 7 张牌放置在 3 列的中间（如观众指出自己记的牌在第 3 列时，收牌时就可按第 1、3、2 列的顺序收集，也可按第 2、3、1 的顺序收集。知道这个原理后，我们也可表演这个魔术了，经过观众 3 次回答，就可快速将观众记住的牌放置到第 2 列第 4 张的位置（即 3 列 7 行的中间位置）。

5.3.2　魔术师是怎么猜出来的

我们已经知道这个小魔术的玩法了，可是，为什么经过 4 次发牌就能确定观众所记的扑克牌位于第 2 列第 4 张呢？

其实，这个魔术是利用余数进行分组的一个例子。

在这个魔术中一共使用了 21 张扑克牌，由于分为了 3 列，因此，可以用牌的序号

除以 3，然后根据余数进行分组。这 21 张牌的顺序与其发牌之后成为 3 行 7 列的位置关系如下：

序号	分组计算	列	行
第1张牌	1 ÷ 3=0 … 1	1	1
第2张牌	2 ÷ 3=0 … 2	2	1
第3张牌	3 ÷ 3=1 … 0	3	1
第4张牌	4 ÷ 3=1 … 1	1	2
第5张牌	5 ÷ 3=1 … 2	2	2
第6张牌	6 ÷ 3=2 … 0	3	2
第7张牌	7 ÷ 3=2 … 1	1	3
第8张牌	8 ÷ 3=2 … 2	2	3
第9张牌	9 ÷ 3=3 … 0	3	3
… …		… …	

从上面的计算过程可看出，余数与所在列有关，并且其关系比较简单，可直接按余数进行分组。具体关系如下：

余数	分组(列)
1	1
2	2
0	3

分析上面中的行还可以看出，在分组计算结果中的商可以确定该张牌在其分组（列）中的位置，即在列中处于第几行，其关系如下：

商	余数	在分组中的位置(行)
0	1	1 (=0+1)
0	2	1 (=0+1)
1	0	1 (=1)
1	1	2 (=1+1)
1	2	2 (=1+1)

2	0	2 (=2)
……		……
n	1	n+1
n	2	n+1
n	0	n

可以看出，当余数为 0 时，所得的商 n 就是该张牌在其分组中的行数 n；若余数不等于 0，所得的商 n 加 1 就是该张牌在其分组中的行数。

我们将牌的序号与其分组位置进行了分析，下面结合魔术师发牌的过程进行运算。

如图 5-8 所示，第一次魔术师在桌面上发了 3 列牌，让观众从中记住一张，只说出该张牌所在的列号，然后将该列牌放在中间。这样，该列的 7 张牌在牌堆中的序号就是 9～14 了，则魔术师第二次发牌时这 7 张牌的位置关系如下：

序号	分组计算	列	行
8	8 ÷ 3=2 … 2	2	3
9	9 ÷ 3=3 … 0	3	3
10	10 ÷ 3=3 … 1	1	4
11	11 ÷ 3=3 … 2	2	4
12	12 ÷ 3=4 … 0	3	4
13	13 ÷ 3=4 … 1	1	5
14	14 ÷ 3=4 … 2	2	5

从上面的计算可看出，魔术师第二轮发牌后，位于第 2 列（原在第 1 列，在收集时按第 2、1、3 列的顺序收牌，就放置在第 2 列的位置）的 7 张牌会分散在 3 列的第 3、4、5 行中。

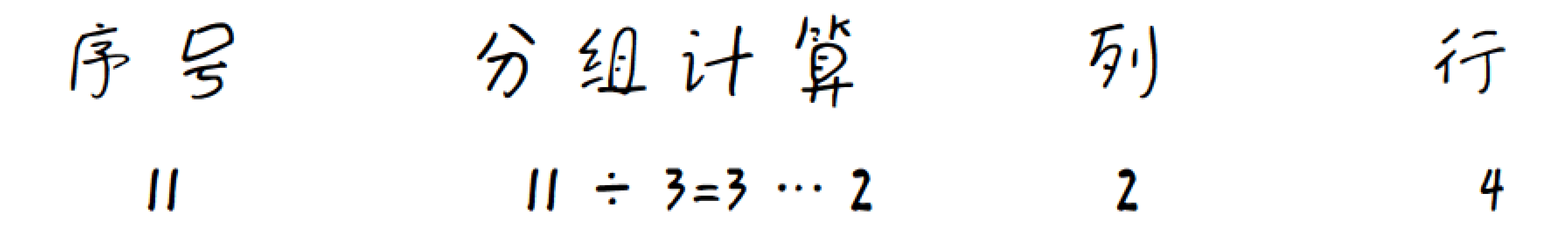
这时，观众所记牌仍然分布在 3 列中的某一列，但位置仍然不清，可以分为 3 种情况：

- 第一种情况，若观众所记牌在第 1 列中，则这些牌位于第 1 列第 4、5 行。
- 第二种情况，若观众所记牌在第 2 列中，则这些牌位于第 2 列第 3、4、5 行。
- 第三种情况，若观众所记牌在第 3 列中，则这些牌位于第 3 列第 3、4 行。

这里以第二种情况来讨论，若观众说明其所记的牌位于第 2 列，根据魔术师收集牌的顺序按第 1、2、3 列收集，则第 2 列第 3、4、5 行这 3 张牌在牌堆中的顺序为 10、11、12。这时魔术师再次进行发牌，则第 10、11、12 张牌所处的行、列位置为：

序号	分组计算	列	行
10	10 ÷ 3=3 … 1	1	4
11	11 ÷ 3=3 … 2	2	4
12	12 ÷ 3=4 … 0	3	4

可以看出，第 10、11、12 张牌经过魔术师重新发牌后，将分发在第 1、2、3 列的第 4 行中。可看出，经过这次发牌，观众所记的牌已分布在各列的第 4 行中了。由于分布在 3 列的第 4 行中，无论观众记住的是哪一列第 4 行中的数据，将该列作为第 2 列收集，这张牌在牌堆中的序号都为 11，则魔术师再次发牌时，序号 11 的牌都将分发到第 2 列第 4 行。



序号	分组计算	列	行
11	11 ÷ 3=3 … 2	2	4

将各种情况绘制一个分解图，如图 5-14 所示。

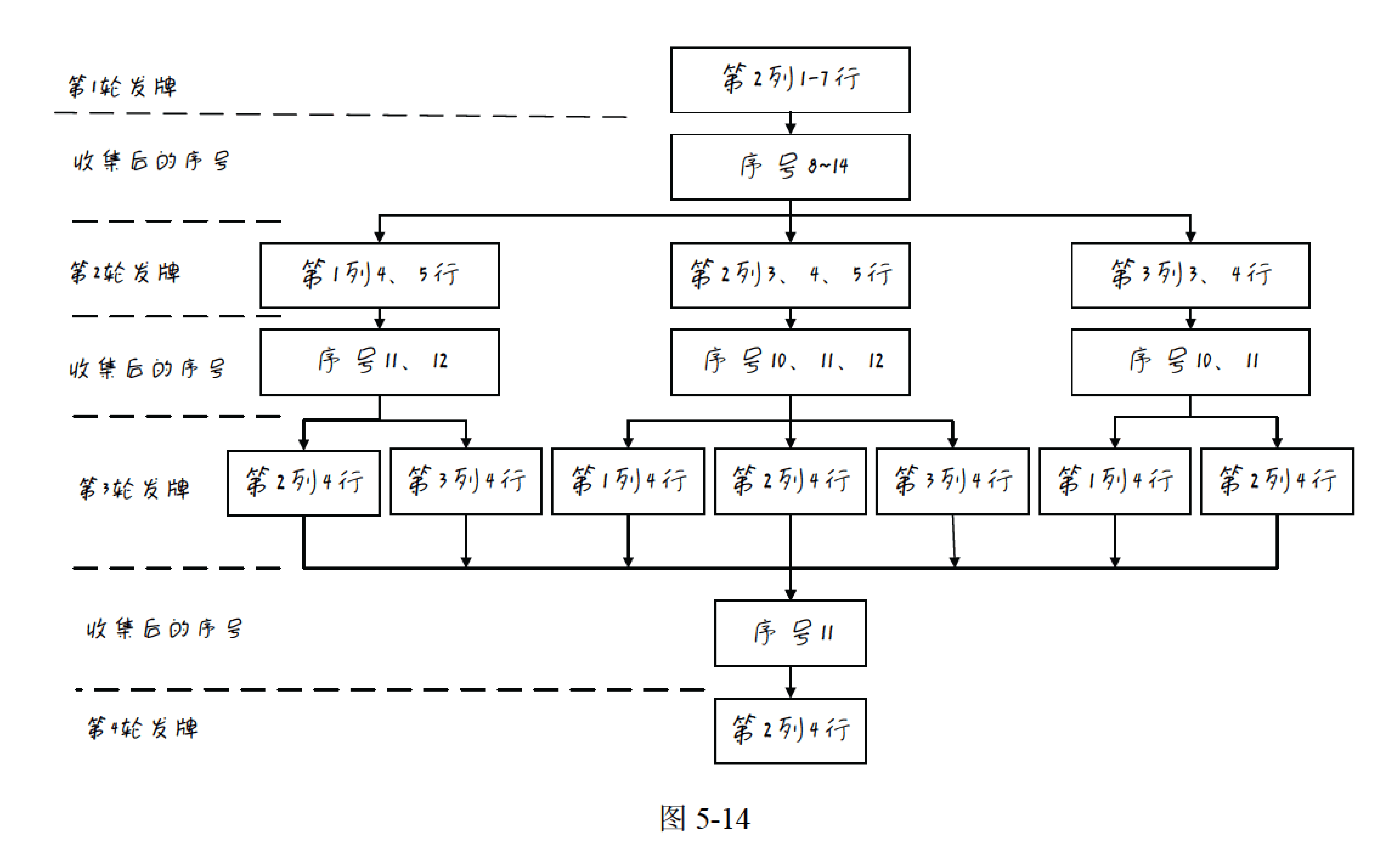


图 5-14

从图 5-14 中可以看出，将观众记住的牌收集到第 2 列（牌堆中的序号为 8～14），无论这张牌在第 1 列、第 2 列还是第 3 列，经过观众 3 次说明其所在列，最终都可将其转到序号 11 处。这时，观众所记的牌已位于牌堆的中间位置，魔术师第 4 次发牌后，就可从第 2 列第 4 行中挑出这张牌。

5.4　奇偶校验

奇偶校验是计算机中最常用的一种验证数据的方法。由于它很简单，所以奇偶校验位用于许多计算机硬件中，当遇到麻烦时能够重新操作，或者通过简单的错误检测就能起到很大作用。例如 SCSI 总线使用奇偶校验位检测传输错误，许多微处理器的指令高速缓存中也包括奇偶校验位保护。因为指令缓存数据是主内存数据的副本，所以在发现错误的时候能够抛弃错误数据并且重新取回数据。

那么奇偶校验是什么原理呢？我们下面来研究一下。

5.4.1　不可靠的网络传输

现在我们天天都在使用网络传输数据，怎样确保传输的数据没有错误呢？

我们都知道，由于网络的情况复杂性，无论是有线、无线传输，都不能保证每次发送的数据会完整无误地传送到对方。

在网络中传送的都是二进制信息，可能会由于线路问题或外界电磁干扰等因素，使传送的二进制位出现错误，如将“1”错误传输为“0”，或将“0”传输为“1”。这种情况，我们称为出现了“误码”。

例如，通过网络传输一个字符 E，这时就将这个字符的 ASCII 传送给对方，字符 E 的 ASCII 码为 69，转化为 2 进制为：

字符“E”的 ASCII 码：$(69)_{10}=(0100\ \ 0101)_2$

由于存在误码，可能会将二进制数据中的一个“0”传为了“1”，得到以下二进制（从右向左第 5 位出了错误）：

$$(0101\ \ 0101)_2=(85)_{10}$$

这时，本来传输的字符 E 就变成了字符 U（字符 U 的 ASCII 码为 85），传输就出错了。

5.4.2　用奇偶校验检查错误

既然网络中传输数据时容易出现误码，那么，该如何发现数据传输中的错误？最简单的检错方法是“奇偶校验”。

奇偶校验方法，是根据被传输的一组二进制代码的数位中“1”的个数是奇数或偶数来进行校验。采用奇数的称为奇校验，反之称为偶校验。

在传送 ASCII 码字符时，使用这种奇偶校验是非常方便的，因为 ASCII 码中字符只占用了低 7 位的二进制，最高位的二进制一直为 0。这时，可将这一位用来保存奇偶

校验。

例如，若采用奇校验，当传输字符 E 时，由于该字符的 ASCII 码的二进制数中包含有 3 个二进制“1”，因此，最高位不用设置，仍然为“0”，使传输的 8 位二进制中“1”的数量为奇数。

若传输字符 F 时，由于该字符的 ASCII 码的二进制为：

$$字符“G”的\ ASCII\ 码：(0100\ \ 0111)_2=(71)_{10}$$

字符 G 的二进制中有 4 个“1”，为偶数，在奇校验时，将这 8 位二进制中的最高位置为 1，如下所示：

$$(1100\ \ 0111)_2=(199)_{10}$$

有了这个奇偶校验，接收方判断接收数据是否出错时就很简单了。例如，若接收方收到以下二进制：

$$(0101\ \ 0101)_2=(85)_{10}$$

可看出，二进制数据中“1”的个数为 4，是偶数，说明接收的数据有误。

如果接收方接收到的数据中“1”的个数为奇数，说明接收的数据是正确的。这时，再将最高位置为 0（即去掉最高位的奇校验数据），则可得到正确的数据。

上面的例子是采用奇校验时的情况，若采用偶校验，其校验方法类似，只是判断二进制数据中“1”的个数是否为偶数。至于采用奇校验还是采用偶校验，是由传输方和接收方事先规定好的。

可以看出，奇偶校验能够检测出信息传输过程中的部分误码，但是只能检查出一位误码，若在一个字节的传输过程中有两位及两位以上误码，就不能检验出了。

另外，奇偶校验只能用来发现错误但不能纠错。发现错误后，只能要求发送方重发。

可以看出，奇偶校验很简单，因此，在很多场合得到了广泛的使用，如前面说的网络传输数据。另外，在内存、硬盘保存数据等方面也得到了广泛应用。

在串行通信中，为了提高效率，奇偶校验位通常是由 UART 这样的接口硬件生成、校验的，在接收方，通过接口硬件中的寄存器的状态位传给 CPU 及操作系统。

错误数据的恢复通常是通过重新发送数据，这个过程通常由如操作系统输入输出程序这样的软件处理的。

5.5 吕洞宾不能坐首位

循环排列位置是数学中一个有趣的问题，也是一个经典问题，可通过求余运算找出规律。下面我们从一个民间传说故事来研究这个问题。

5.5.1　座位安排

传说铁拐李、汉钟离、张果老、蓝采和、何仙姑、吕洞宾、韩湘子、曹国舅等 8 人称为八仙（如图 5-15 所示）。这天，八仙到天宫拜会王母娘娘。王母娘娘很高兴地地接待他们，正要让他们在一张八仙桌旁坐下时，有一件事让王母十分为难。按礼节她应该让八仙之首先入席，坐首位。可是八仙之首究竟该是谁呢？好像没有排出谁是老大。

这时王母的香案使（王母身边的工作人员）想出了一个主意，他对八仙说："我对各位一视同仁，毫无偏见。就由我来安排你们的座位吧，你们先排成一个圆圈，我请王母来掷两粒骰子，看看共掷出几点，就按这个点数，从第一个人开始数起，依次数到这个点数时，这个人就排除在外。这样周而复始，最后留下谁，谁就是八仙之首，让他先入席坐首位。"

八仙一听这办法有趣，也很公平，立即赞成，王母娘娘也觉得这个主意不错，也点头同意。

就在这时，观音菩萨也来了。观音看不惯吕洞宾的一些行为，不想让他成为八仙之首，便在王母耳边嘀咕了一番。

王母笑道："菩萨放心，骰子掷出的点数纯属偶然，他只有八分之一的机会，未必能当上八仙之首。"

观音摇头说："不行，我一定要排除他！"

王母听了此言，甚觉为难。观音说："我有一个补救的方法，反正位置由香案使安排，我告诉他，把吕洞宾安排在某一位置上，然后从这个位置按顺时针方向计数，他就无法做八仙之首了。"

王母大喜，叫过香案使，观音对其耳语几句。然后，香案使按观音的指点，请八仙依次排成一圈，开始掷骰子数点数。后来，吕洞宾果真未当上八仙之首。



图 5-15

那么，香案使究竟把吕洞宾安排在哪个位置，才能确保将他排除出去呢？

5.5.2 试排座位找规律

接下来，我们来进行一下试排座位。

再来看一下香案使说的规则：按掷出骰子的点数，从第一个人开始数起，依次数到这个点数时，这个人就排除在外。这样周而复始，最后留下谁，谁就是八仙之首，让他先入席坐首位。

可以看出，需要解决两个问题：

首先要确定第一个人是谁，是从哪一个位置开始数起。

还需要确定掷出骰子的点数。由于是两粒骰子，点数最小为 2（两粒都为 1 点时），最大为 12（两粒都为 6 点时）。

其实，第一个人是谁不是大问题，排在第一位也可能会被排除。因此，只要将八仙进行编号，然后围成一个圈就可以，假如分别编号为 X1、X2、X3、……、X7、X8，如图 5-16 所示。

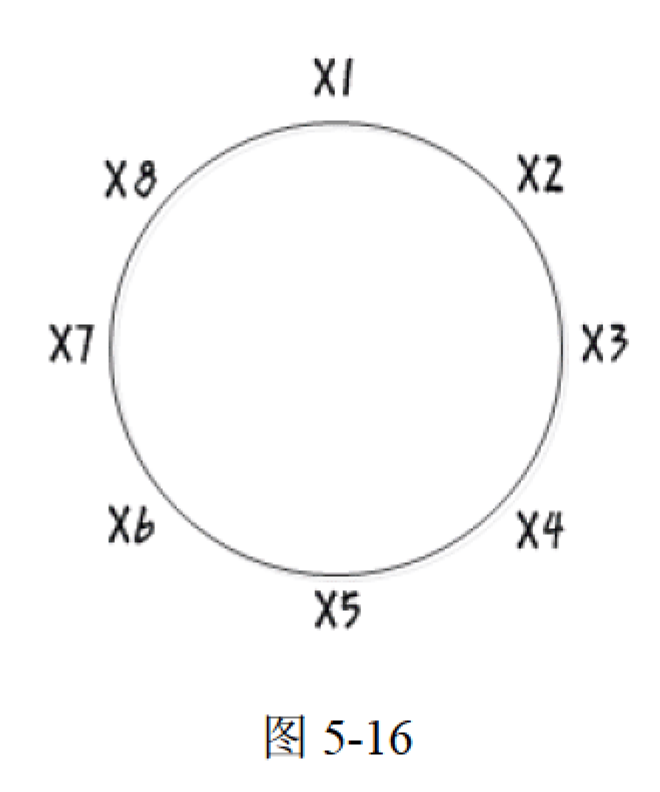


图 5-16

如果骰子点数为 2，根据图 5-16 所示排列序号，被排除的顺序依次为：

最初顺序：	X1、	X2、	X3、	X4、	X5、	X6、	X7、	X8
点数排除：	1	2						
剩下顺序：	X1、		X3、	X4、	X5、	X6、	X7、	X8
点数排除：			1	2				
剩下顺序：	X1、		X3、		X5、	X6、	X7、	X8
点数排除：					1	2		
剩下顺序：	X1、		X3、		X5、		X7、	X8
点数排除：							1	2
剩下顺序：	X1、		X3、		X5、		X7	

点数排除：	1		2					
剩下顺序：	X1、				X5、		X7	
点数排除：					1		2	
剩下顺序：	X1、				X5			
点数排除：	1				2			
剩下顺序：	X1							

最后只剩下一个序号 X1，表示排在第一个位置的最后被保留，未被排除，则排在第一个位置的为八仙之首。因此，吕洞宾不能安排在第一个位置。

那么，如果骰子点数为 3，被排除的顺序又是什么样的呢？

最初顺序：	X1、	X2、	X3、	X4、	X5、	X6、	X7、	X8
点数排除：	1	2	3					
剩下顺序：	X1、	X2、		X4、	X5、	X6、	X7、	X8
点数排除：				1	2	3		
剩下顺序：	X1、	X2、		X4、	X5、		X7、	X8
点数排除：	3						1	2
剩下顺序：		X2、		X4、	X5、		X7、	X8
点数排除：		1		2	3			
剩下顺序：		X2、		X4、			X7、	X8
点数排除：		3					1	2
剩下顺序：				X4、			X7、	X8
点数排除：				1			2	3
剩下顺序：				X4、			X7	
点数排除：				1(3)			2	
剩下顺序：							X7	

最后剩下的序号是 X7，因此，吕洞宾也不能排在第 7 个位置上，否则也可能成为八仙之首。

按上面的方法，继续分析当骰子点数为 4～12 时的情况。最后得出骰子点数从 2～12 点时，各位置被排除的先后顺序如下（最后一个序号就是剩下的）：

2 点时：X2、X4、X6、X8、X3、X7、X5、X1

3 点时：X3、X6、X1、X5、X2、X8、X4、X7

4 点时：X4、X8、X5、X2、X1、X3、X7、X6

5点时：X5、X2、X8、X7、X1、X4、X6、X3

6点时：X6、X4、X3、X5、X8、X7、X2、X1

7点时：X7、X6、X8、X2、X5、X1、X3、X4

8点时：X8、X1、X3、X6、X5、X2、X7、X4

9点时：X1、X3、X6、X4、X5、X2、X7、X8

10点时：X2、X5、X1、X8、X4、X6、X3、X7

11点时：X3、X7、X5、X6、X2、X8、X1、X4

12点时：X4、X1、X8、X3、X2、X7、X6、X5

对最后的序号进行总结，在 11 种骰子点数情况下，各位置被保留下来的次数如下：

X1：2次

X2：0次

X3：1次

X4：3次

X5：1次

X6：1次

X7：2次

X8：1次

可以看出，第 4 个位置被保留下来的次数为 3 次，概率最大，而第 2 个位置一次都没有被保留下来。

因此，观音对香案使的指点就是：将吕洞宾排在第 2 个位置，这样就可以确保他被排除。

5.5.3 西方的约瑟夫环

其实，对于循环排座位的问题，西方也有一个类似的故事。

据说著名历史学家 Josephus（约瑟夫）经历过以下故事：在罗马人占领乔塔帕特后，40 个犹太人和 Josephus 躲在一个山洞中。40 个犹太人决定宁死也不被敌人抓到，于是决定集体自杀。大家经过讨论决定了一个自杀方式，41 个人围成一个圆圈，由第 1 个人开始报数，每报数到 3 的人就必须自杀，然由再由下一个人重新开始报数，直到所有人都自杀身亡为止。

然而 Josephus 并不想遵从这个规则，不想自杀。于是，Josephus 先假装同意该方案，然后坐到大家围成圆圈的第 31 个位置，最后逃过了这场死亡游戏。

那么，为什么 Josephus 坐在第 31 位置就可逃过该死亡游戏呢？

首先将 41 个人排成一个圆圈，并编好序号，如图 5-17 中圆内的编号（圆圈内的编号是座位序号，圆圈外的数字是每个人报到 3 的顺序）。然后从编号 1 的人开始报数，报到 3 就表示该人是第 1 个该自杀的人，序号为 3 的人首先数到 3（将序号为 3 的人排除到圆圈之外），接下来序号为 4 的人又从 1 开始报数，……，这样不停地循环，序号为 31 的人将是最后剩下的一个人。由于前面的人都已自杀，没法监督到最后这个人是否遵守游戏规则了。

在这个故事中，用 41 个序号来表示这些人，不再是八仙中的 8 个序号，如果手工推算，需要很长的时间。因此，我们可以考虑借助计算机程序来进行推算。

在设计程序时，我们还可以考虑参与循环点数的人数 *N* 是一个可变的值，可将该数任意扩大或缩小，并假设有 *M* 个朋友不幸要参加这个游戏，又该如何保护自己和朋友，使这 *M* 个朋友都留在最后。

在计算机程序中，可以使用一种叫循环链表的数据结构来模拟约瑟夫环的结构，不过，这需要编写操作循环链表相关的代码，程序的代码比较多。为了简化程序，我们可以考虑用数组来保存约瑟夫环中应该出列的顺序序号数据，而数组的下标作为参与人员的编号，并将数组看作为环形来处理。

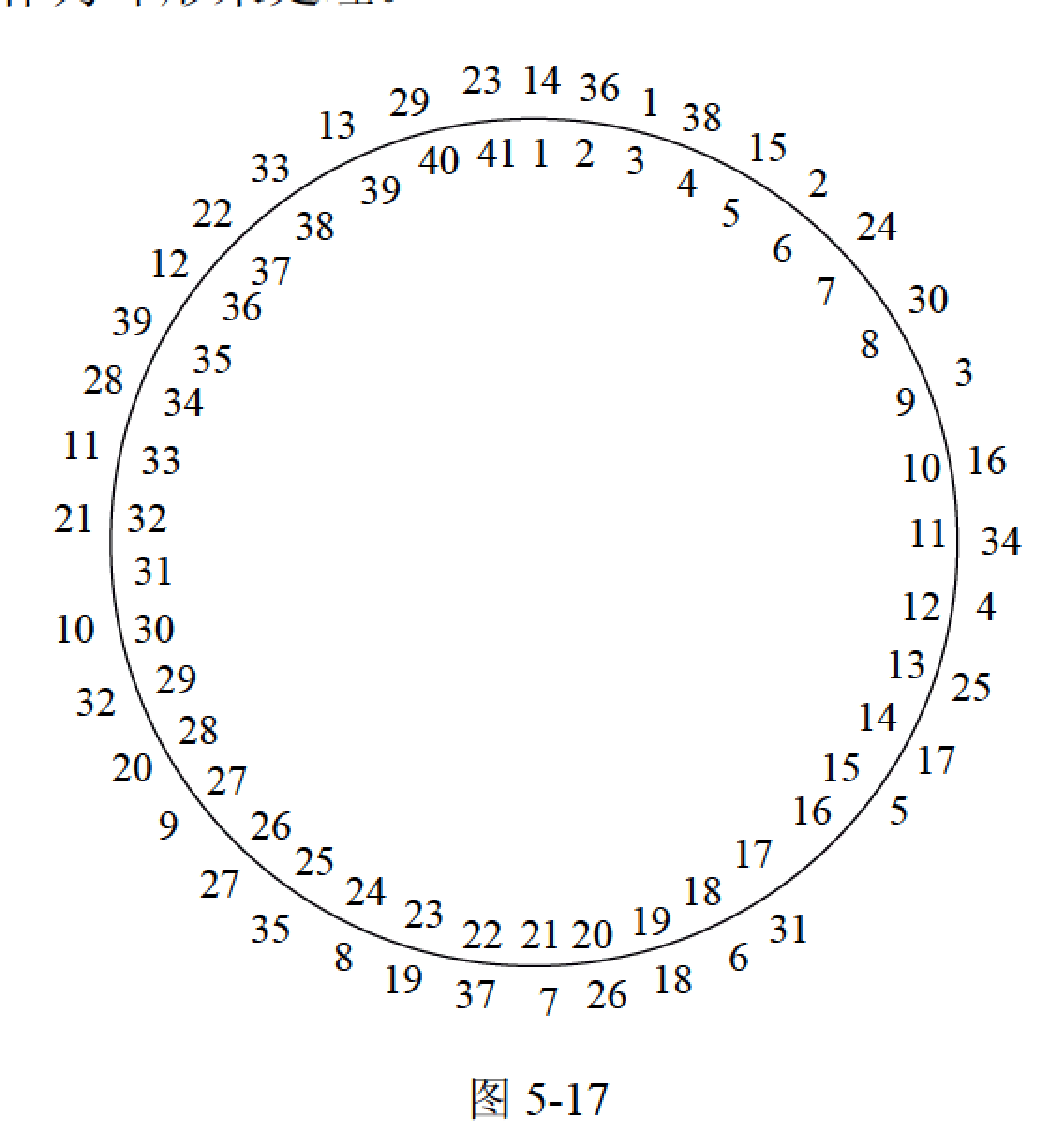


图 5-17

具体的程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#define N 41                  //总人数
#define M 3                   //数到 3 出列
int main()
{
   int man[N]={0};            //保存出列的序号，为 0 表示未出列
   int count=1;               //出列记数器
   int i=0;                   //报数记数器
```

```
    int pos=-1;                 //位置记数器
    int alive=0;
    while(count<=N)             //在 N 个人中模拟循环报数
    {
        do{
            pos=(pos+1) % N;    //求余，进行环状处理
                                //如果记数器超过 N 值，又从 1 开始
            if(man[pos]==0)     //若编号 pos 还未出列
                i++;            //报数
            if(i==M)            //报数 M 的人
            {
                i=0;            //初始化记数器，又从 1 开始报数
                break;
            }
        }while(1);
        man[pos]=count;         //保存出列序号
        count++;
    }
    printf("\n 约瑟夫排列(最初位置-约瑟夫环位置):\n");    //输出排列位置
    for(i=0;i<N;i++)
    {
        printf("%d-%d  ",i+1,man[i]);
        if(i!=0 && i%10==0)     //每输出 10 个则换行
            printf("\n");
    }
    for(i=0;i<N;i++)            //查找到需保留的编号
        if(man[i]>N-1)
            printf("\n 初始序号:%d,约瑟夫环序号:%d\n",i+1,man[i]);
    printf("\n");
    getch();
    return 0;
}
```

在上面的程序中，每行主要代码右侧都写有注释。

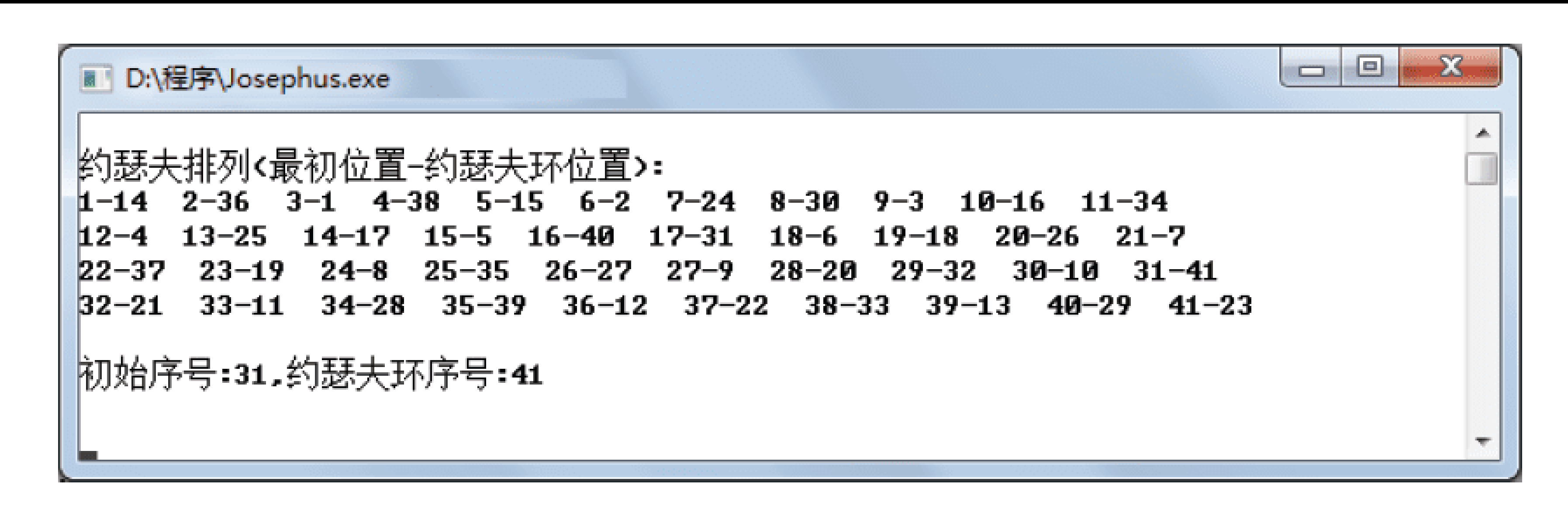
程序的算法也很简单：

（1）用一个数组保存约瑟夫环，其中数组元素的序号为参与人员的初始位置号，每个数组元素的值，就是对应序号人员出列的顺序。例如，若 man[0]=14，表示排在第 1 个位置的人将是第 14 个出列的人（数组元素是从 0 开始的，所以 man[0]中的 0 表示第 1 个人）。

（2）不断循环，模拟报数的过程，将报到 M（这里为 3）的序号 pos 处的数组元素记上其出列的序号 count（即设置 man[pos]=count），这样，man[n]的值若为 0，表示还未出列。直到出列序号 count 达到总参与人数 N 为止。

（3）在 man 数组中已将出列序号标出来了，接下来就是找到最后出列的人，再找出其对应的原始位置序号。只要最初坐在这个序号对应的位置，就能确保其在最后才出列。

编译执行以上程序，程序首先显示出约瑟夫环的数列结果（前一个数据是最初的编号，后一个数据是约瑟夫环中的编号）。接着提示初始号为 31 的位置最后出列，最后出列的序号为 N（41），即排在第 31 号位置的人在第 41 次出列，执行过程如图 5-18 所示。


D:\程序\Josephus.exe

```
约瑟夫排列<最初位置-约瑟夫环位置>:
1-14  2-36  3-1  4-38  5-15  6-2  7-24  8-30  9-3  10-16  11-34
12-4  13-25  14-17  15-5  16-40  17-31  18-6  19-18  20-26  21-7
22-37  23-19  24-8  25-35  26-27  27-9  28-20  29-32  30-10  31-41
32-21  33-11  34-28  35-39  36-12  37-22  38-33  39-13  40-29  41-23

初始序号:31,约瑟夫环序号:41
```

图 5-18

5.5.4　用数学方法解约瑟夫环

上面编写的解约瑟夫环的程序模拟了整个报数的过程，程序运行时间还可以接受，很快就可以出计算结果。可是，当参与的总人数 N 及出列值 M 非常大时，其运算速度就慢下来。例如，当 N 的值有上百万，M 的值为几万时，到最后虽然只剩 2 个人，也需要循环几万次（M 的数量）才能确定 2 个人中下一个出列的序号。显然，在这个程序的执行过程中，很多步骤都是进行重复无用的循环。

那么，能不能设计出更有效率的程序呢？

办法当然有。其中，在约瑟夫环中，只是需要求出最后的一个出列者最初的序号，而不必要去模拟整个报数的过程。因此，为了追求效率，可以考虑从数学角度进行推算，找出规律然后再编写程序即可。

为了讨论方便，先根据原意将问题用数学语言进行描述。

问题：将编号为 0～（N–1）这 N 个人进行圆形排列，按顺时针从 0 开始报数，报到 M–1 的人退出圆形队列，剩下的人继续从 0 开始报数，不断重复。求最后出列者最初在圆形队列中的编号。

下面首先列出 0～（N–1）这 N 个人的原始编号如下：

0　1　2　3　···　N-3　N-2　N-1

根据前面曾经推导的过程可知，第一个出列人的编号一定是（M–1）%n。例如，在 41 个人中，若报到 3 的人出列，则第一个出列人的编号一定是（3–1）%41=2，注意这里的编号是从 0 开始的，因此编号 2 实际对应以 1 为起点中的编号 3。根据前面的描述，m 的前一个元素（M–1）已经出列，则出列 1 人后的列表如下：

0　1　···　M-3　M-2　□　M　M+1　M+2　···　N-2　N-1

根据规则，当有人出列之后，下一个位置的人又从 0 开始报数，则以上列表可调整为以下形式（即以 M 位置开始，N–1 之后再接上 0、1、2……，形成环状）：

M　M+1　M+2　···　N-2　N-1　0　1　···　M-3　M-2

按上面排列的顺序重新进行编号，可得到下面的对应关系：

$$
\begin{array}{cccccccccc}
0 & 1 & 2 & & \cdots & \cdots & & & N-3 & N-2 \\
M & M+1 & M+2 & \cdots & N-2 & N-1 & 0 & 1 & \cdots & M-3 & M-2
\end{array}
$$

即，将出列 1 人后的数据重新组织成了 0～（*N*–2）共 *N*–1 个人的列表，继续求 n–1 个参与人员，按报数到 *M*–1 即出列，求解最后一个出列者最初在圆形队列中的编号。

看出什么规律没有？对了，通过一次处理，将问题的规模缩小了。即，对于 *N* 个人报数的问题，可以分解为先求解（*N*–1）个人报数的子问题；而对于（*N*–1）个人报数的子问题，又可分解为先求[（*N*–1）–1]人个报数的子问题，……。

问题中的规模最小时是什么情况？就是只有 1 个人时（*N*=1），报数到（*M*–1）的人出列，这时最后出列的是谁？当然只有编号为 0 这个人。因此，可设有以下函数：

$$F(1)=0$$

那么，当 *N*=2，报数到（*M*–1）的人出列，最后出列的人是谁？应该是只有一个人报数时得到的最后出列的序号加上 *M*，因为报到 *M*-1 的人已出列，只有 2 个人，则另一个出列的就是最后出列者，可用公式表示为以下形式：

$$F(2)=F(1)+M$$

通过上面的算式计算时，F(2)的结果可能会超过 *N* 值（人数的总数）。例如，设 *N*=2，*M*=3（即 2 个人，报数到 2 时就出列），则按上式计算得到的值是：

$$F(2)=F(1)+M=0+3=3$$

一共只有 2 人参与，编号为 3 的人显然没有。怎么办？由于是环状报数，因此当两个人报完数之后，又从编号为 0 的人开始接着报数。根据这个原理，即可对求得的值与总人数 *N* 进行模运算，即：

$$F(2)=[F(1)+M]\%2$$

验证一下：

$$
\begin{aligned}
F(2)&=[F(1)+M]\%2\\
&=[(0+3)]\%2\\
&=1
\end{aligned}
$$

即，*N*=2，*M*=3（即有 2 个人，报数到 3–1 的人出列）时，循环报数最后一个出列的人的编号为 1（编号从 0 开始）。我们来推算一下，如下所示，当编号为 0、1 的两个人循环报数时，编号为 0 的人报的数为 0 和 2，当报到 2（*M*–1）时，编号 0 出列，最后剩下编号为 1 的人，所以编号为 1 的人最后出列。

编号：0　　1

报数：0(2)　　1

根据上面的推导过程，可以很容易推导出，当 N=3 时的公式：

$$F(3)=[F(2)+M]\%3$$

同理，也可以推导出参与人数为 N 时，最后出列人员编号的公式：

$$F(N)=[F(N-1)+M]\%N$$

其实，这就是一个递推公式，公式包含以下两个式子：

$$F(1)=0$$

$$F(N)=[F(N-1)+M]\%n \qquad (N>1)$$

有了这个递推公式，再来设计程序就很简单了，可以用递归的方法来设计程序，具体代码如下：

```
#include <stdio.h>
int main(void)
{
    int n,m,i,s=0;
    printf ("输入参与人数 N 和出列位置 M 的值 = ");
    scanf("%d%d",&n,&m);

    printf ("最后出列的人最初位置是 %d\n",josephus(n,m));
    getch();
    return 0 ;
}

int josephus(int n,int m)
{
    if(n==1)
        return 0;
    else
        return (josephus(n-1,m)+m)%n;
}
```

在以上代码中，定义了一个递归函数 josephus()，然后在主函数中调用这个函数进行运算。

编译执行以上程序，输入 N 和 M 的值，可以很快得到最后出列人的编号，输入 N=8，M=3，得到的结果如图 5-19 所示（注意编号是从 0 开始）。

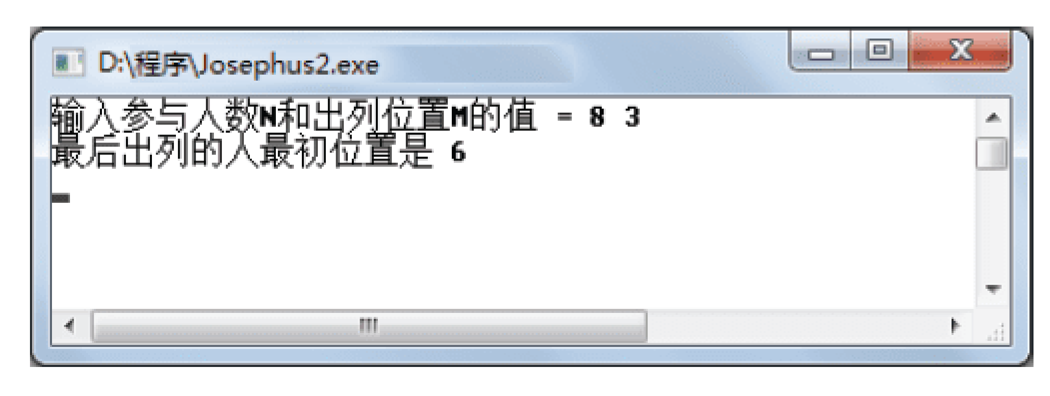


图 5-19

使用递归函数会占用计算机较多的内存，当递归层次太深时可能导致程序不能执行，因此，也可以将程序直接编写为以下的递推形式：

```
#include <stdio.h>
int main(void)
{
   int n,m,i,s=0;
   printf ("输入参与人数 N 和出列位置 M 的值 = ");
   scanf("%d%d",&n,&m);
   for (i=2; i<=n; i++)
      s=(s+m)%i;
   printf ("最后出列的人最初位置是 %d\n",s);
   getch();
   return 0 ;
}
```

这段代码执行的结果与递归程序执行结果完全相同。

可以看出，经过一些数学推导，最后总结出规律简化程序，将几十行的代码缩减到几行。更主要的是，程序执行的效率得到大大的提升，省去了很多重复的循环，既使求解的 N 和 M 值很大，也不会成为问题。

5.6 智叟分牛

余数在我们生活中无处不在，特别是在平均分配某些不可分割的物体时，很多时候都会有余数出现。例如，要将 11 台电视机中的一半留下来。这时，由于 11 不能被 2 整除，而一台电视机又不能切割为两部分，就会有余数产生。

对于这种经过分配后会产生余数的情况，比较有趣的一个例子是：智叟分牛，本节我们来看看智叟分牛的解决方案。

5.6.1 遗产分配难题

传说古代印度有一位老人，临终前留下遗嘱，要把所养的 19 头牛分给 3 个儿子。其中，老大分得总数的二分之一、老二分得总数的四分之一、老三分得总数的五分之一。按印度的教规，牛被视为神灵，不能被宰杀，只能按整头数分配。父亲的遗嘱更需要无条件遵守。老人死后，兄弟三人对分牛的事一筹莫展。

按老人的遗嘱，老大应分得的牛的数量为：

$$19 \div 2=9.5$$

根据分得总数的二分之一来计算，得到的不是整数的头牛，而是 9.5 头，显然，有小数部分就必须要分割一头牛。而根据教规，牛是不能被宰杀分割的！

同样，老二分得总数的四分之一，应分得的牛的数量为：

$$19 \div 4=4.75$$

也有小数！

同样，老三分得总数的五分之一，应分得的牛的数量为：

$$19 \div 5=3.8$$

也有小数！

该怎么分配才既符合本国的教规，又能遵守父亲的遗嘱呢？兄弟三人始终想不出好的办法来。

5.6.2　智叟给出的分配方案

正在三兄弟愁眉不展，想不出分配办法时，智叟牵着一头牛从三兄弟家门前经过，于是，三兄弟赶紧向智叟请教方法。

听了三兄弟说出原委后，智叟说：这好办。我把这头牛借给你们。这样，共有 20 头牛，就好分配了。分完之后，再把借我的这一头牛还给我就行了。

果然，牛的总数从 19 变为 20 后，老大分得二分之一，得到的牛的数量为：

$$20 \div 2=10$$

同样，老二分得总数的四分之一，应分得的牛的数量为：

$$20 \div 4=5$$

老三分得总数的五分之一，应分得的牛的数量为：

$$20 \div 5=4$$

三兄弟共分配的牛的总数为：

$$10+5+4=19$$

而借了智叟的一头牛后，牛的总数是 20，三兄弟分了 19 头，正好剩下一头，还给智叟。这样，三兄弟每人分得的牛的头数为整数，也按父亲的遗嘱比例进行了分配。

5.6.3　分配原理

可是，为什么智叟在牛的总数中加上 1 头牛，然后就可完成这种分配呢？19 头牛中的一半为什么是 10 头呢？

下面我们用算式来计算一下。

按老人的遗嘱，19 头牛按老大得到二分之一、老二得到四分之一、老三得到五分之一，则三个人应分得 $\frac{19}{2}$ 头、$\frac{19}{4}$ 头和 $\frac{19}{5}$ 头。将三兄弟分得牛的数量相加：

$$\frac{19}{2}+\frac{19}{4}+\frac{19}{5}=\frac{361}{20}$$

可以看出，三兄弟并没有将牛分完，还剩下：

$$19-\frac{361}{20}=\frac{19}{20}$$

那么，根据父亲的遗嘱，剩下的部分也需要按二分之一、四分之一、五分之一进行分配，则老大还可分得：

$$\frac{19}{20}\div 2=\frac{19}{20\times 2}$$

老二还可分得：

$$\frac{19}{20}\div 4=\frac{19}{20\times 4}$$

老三还可分得：

$$\frac{19}{20}\div 5=\frac{19}{20\times 5}$$

并且，经过第二次分配之后，牛还是未被分配完，还会剩下：

$$\frac{19}{20\times 20}=\frac{19}{20^2}$$

对剩下的这部分，还需要按老大、老二、老三各自的比例进行分配。

可以看出，这种分配是可以无限进行下去的，因此，老大分得的牛的总数应该为：

$$19\times\frac{1}{2}+\frac{19}{20^1}\times\frac{1}{2}+\frac{19}{20^2}\times\frac{1}{2}+\frac{19}{20^3}\times\frac{1}{2}+\cdots$$

类似地，老二分得的牛的总数应该为：

$$19\times\frac{1}{4}+\frac{19}{20^1}\times\frac{1}{4}+\frac{19}{20^2}\times\frac{1}{4}+\frac{19}{20^3}\times\frac{1}{4}+\cdots$$

同样，老三分得的牛的总数应该为：

$$19\times\frac{1}{5}+\frac{19}{20^1}\times\frac{1}{5}+\frac{19}{20^2}\times\frac{1}{5}+\frac{19}{20^3}\times\frac{1}{5}+\cdots$$

可以看出，三兄弟分得的牛的总数构成了 3 个无穷递缩等比数列，分别对这 3 个等

比数列求和，可知老大、老二、老三分得的牛的总数分别为 10、5、4 头。正好是智叟分牛的结果。

对无穷递缩等比数列求和，需要用到高中的数学知识。下面我们再来看另一种分析方法比例法。

在本题中，由于要将 19 头牛按一定比例完全分配给三兄弟，则三兄弟分得牛的数量的比例为：

$$\frac{1}{2}:\frac{1}{4}:\frac{1}{5}=10:5:4$$

也就是说，老大应从牛的总数中分得的比例为：

$$\frac{10}{10+5+4}=\frac{10}{19}$$

类似地，老二应从牛的总数中分得的比例为：

$$\frac{5}{10+5+4}=\frac{5}{19}$$

同样，老三应从牛的总数中分得的比例为：

$$\frac{4}{10+5+4}=\frac{4}{19}$$

可以看出，通过比例法计算出来的结果中三人分得牛的数量分别为 10、5、4，也正好将全部的牛分完。

第 6 章　概率——你运气好吗

本次列车能正点到达吗？

这次谈判的项目，有几成把握能签下来？

这个月我申请的机动车牌照号能摇中吗？

我买的彩票能中奖吗？

在现实生活中，我们常常会遇到这类问题。对于这些问题，平常我们总是将其归结为是否有运气。运气好时就能签下合同、摇中车牌、中彩票……。

其实，从数学角度来看，这些都是与概率相关的问题。本章我们来研究概率相关的问题。

6.1　初中学习过的概率

作为程序员，应该对数学中的概率有一定了解，在程序设计的很多地方都可能会用到概率知识。例如，对于一些复杂知识库的判别，就可以用概率。

6.1.1　谁先开球

先来看一个场景。

在足球比赛中，两支球队中只能有一支球队先开球，哪一队先开球就占有先进攻的优势。另外，球场中的阳光、风向也对比赛有重要影响。因此，为了公平，通常会要求两队中的一队选择球门，另一队选择先开球。这时裁判应该怎么决定哪队选球门，哪队先开球呢？

足球比赛之前，裁判员通常用抛硬币的方法让比赛双方的队长猜硬币的正反面，猜中正面的这一队选球门，另一队则先开球。

为什么选用抛硬币的方法来决定呢？

因为大家都觉得这种方法很公平，能保证两队选门或开球的可能性一样大。

那么，为什么大家会觉得这种抛硬币的方法很公平呢？如图 6-1 所示，可看到硬币有正反两面。在抛硬币时，事先不能确定是正面向上还是反面向上，但是，参赛双方很容易感觉到正面向上或反面向上的可能性是一样的，因此，两队获得选门或开球的可能性是一样的。

对于这种可能性，就称为概率，也称为或然率、机会率或机率。使用概率可对随机

事件发生的可能性进行度量。



图 6-1

例如，在上面的例子中，两支球队的队长猜中硬币正面的概率各为 50%时，机会均等，因此大家都会觉得这是公平的。

类似的例子还有很多。

例如，在围棋比赛中如何确定参赛双方谁下黑子（如图 6-2 所示）？在围棋中不是采用猜硬币正面的方法，而是采用另一种方法，猜围棋子的单双。在围棋中的术语称为“猜先”，即猜对了的就先下子（下黑子）。猜先的具体作法是：比赛双方中的一方（通常是段位高或年长者）抓一把白棋放在棋盘上，另一方猜对方抓的这一把棋子为单数还是双数，猜对了就可自己选择下黑子还是下白子，若未猜对，则由对方选择。

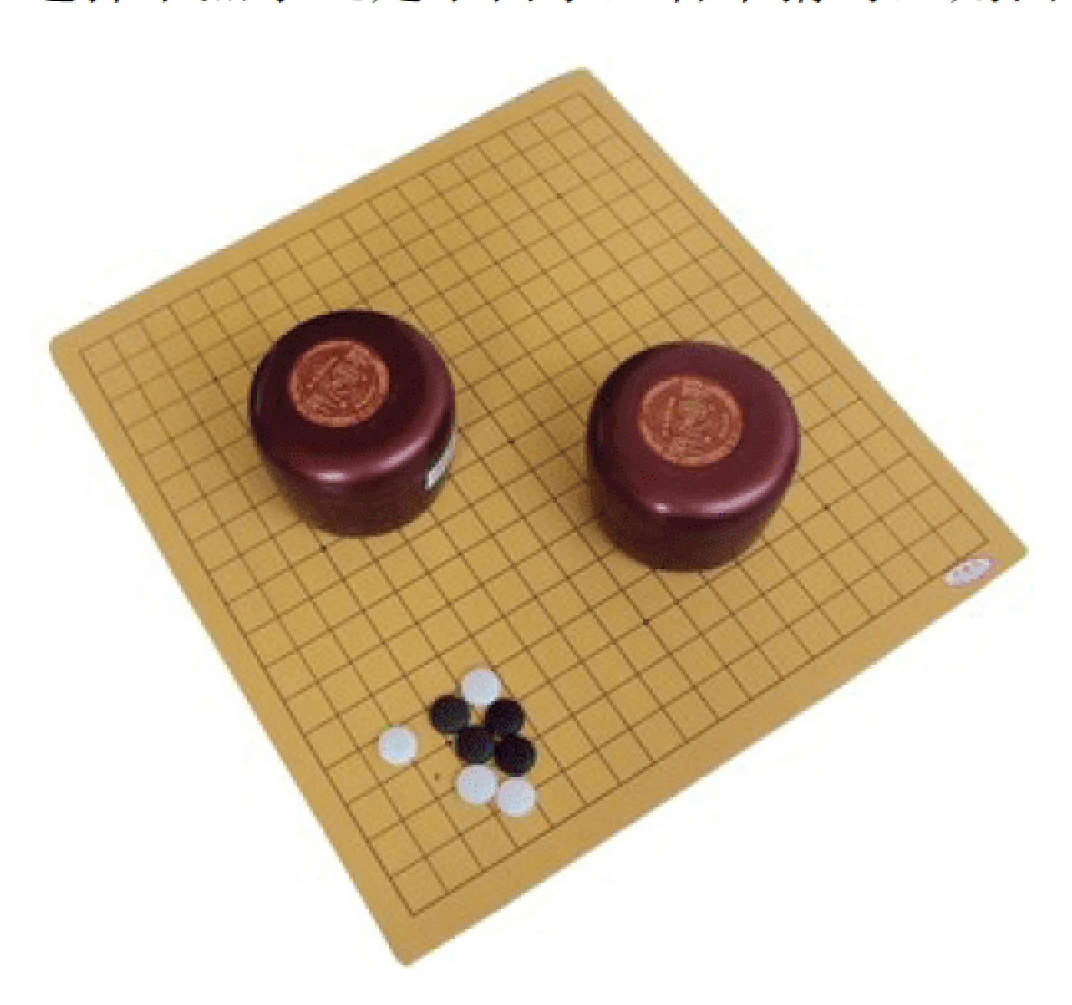


图 6-2

在围棋的“猜先”中，参赛一方所抓围棋子只有两种情况：要么为单数、要么为双数(这就和硬币可以有正反面两种情况类似)。事先双方都不知道棋子的数量是单还是双，为单数或双数的概率是相等的，都为 50%。因此，这种方法感觉是公平的。

6.1.2　用程序模拟抛硬币

那么，在抛硬币、猜单双时，硬币的正反面（或棋子的单双）出现次数是否真的完

全一致呢？对于这一点，我们可以做一个实验，通过多次抛硬币，记录其正反面出现的次数，然后计算出现的频率。

首先，制作如表 6-1 所示的记录表格。

表 6-1　硬币正面向上次数

总次数（n）	100	200	300	400	500	……
正面向上（m）						
正面向上频率（m/n）						

接着开始按表格中的“总次数”开始抛掷硬币，并将正面向上的次数记录到表格中。

如果要手工去完成表 6-1 中的统计数据，需要较长的时间。幸好，我们是程序员，可以考虑使用计算机程序来解决问题。

在 C 语言中，有一个随机函数 rand()，可产生在 0～RAND_MAX 范围内的一个随机数，即，rand()函数生成的这个数是未知的。我们可以使用这个函数来模拟抛掷硬币的情况。

对于通过 rand()函数得到的一个随机数，怎么将其对应于硬币的正、反两面呢？

一种方法是对 rand()函数生成的随机数进行运算，如下式：

(int)(2*rand() / (RAND_MAX+1))

这样，最终得到的结果只能为 0 和 1 这两种情况，假设结果为 1 时代表硬币正面向上，则只需要统计结果为 1 的次数，就可得到模拟硬币正面向上的次数了。

另一种更简洁的方法，就是对 rand()函数生成的随机数进行奇偶判断（类似于围棋中的猜单双）。假设为奇数时表示硬币正面向上，则只需要统计 rand()函数生成的随机数为奇数时的次数，就可得到模拟硬币正面向上的次数了。

根据以上思路编写 C 语言程序如下：

```
#include <stdio.h>

int main()
{
    int i,m=0,n=0;                //m 表示正面向上的次数，n 表示总次数

    srand((int)time(0));          //设置随机数种子
    printf("输入抛硬币的次数：");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        if(rand()%2==1)           //正面向上
            m++;
    }

    printf("\n 抛掷 %d 次硬币的统计数据：\n",n);
    printf("硬币正面向上的次数:%d,频率:%.2f\n",m,(float)m/n);

    getch();
    return 0;
}
```

运行以上程序，输入抛掷硬币的次数为 100，得到如图 6-3 所示的结果。由运算结果可看出，正面向上的频率并不是一半（0.5），而是 0.49。并且，如果再次运行程序，同样输入抛掷硬币的次数为 100，得到的频率不一定仍是 0.49，可能是 0.48、0.49、0.51、0.52 等值。

将获得的“正面向上的频率”填入表格中。多次运行程序，分别输入不同的抛掷硬币的次数，将计算出来的“正面向上频率”数据填入表格中，得到如表 6-2 所示的数据。



图 6-3

表 6-2　硬币正面向上次数

总次数（n）	100	500	1000	1500	2000	2500
正面向上（m）	49	240	5203	753	989	1226
正面向上频率（m/n）	0.49	0.48	0.52	0.50	0.49	0.49

根据表 6-2 所统计的频率，制作出如图 6-4 所示的折线图。

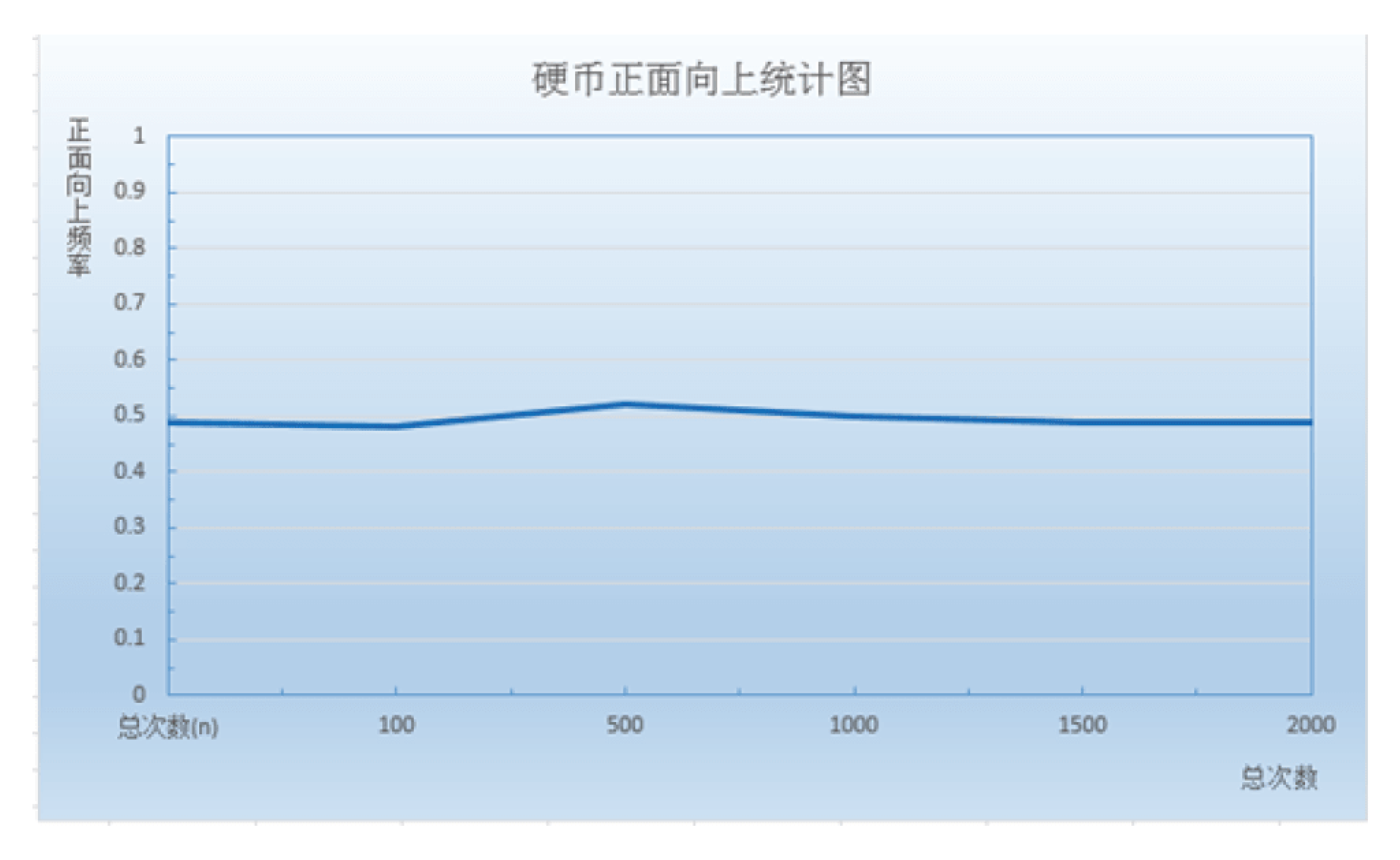


图 6-4

从图 6-4 中可看出硬币“正面向上”的频率变化趋势。根据计算，每一组抛掷硬币中“正面向上”的频率都不相同，即每次试验中随机事件发生的频率具有不确定性。但

是，也可发现随机事件发生的频率也有规律：在试验次数较少时，“正面朝上”的频率变化较大，而随着试验次数的逐渐增加，频率会趋于稳定，且“正面朝上”的频率越来越接近 0.5。因此，这里就用 0.5 这个常数表示“正面朝上”发生的可能性大小。

6.1.3 什么是概率

通过运行上面的程序，并通过计算得出的数据生成图表，可归纳总结出以下结论。

实验证明，随机事件发生的频率逐渐稳定到某一个常数，通常这个常数刻画了随机事件的可能性大小，这个常数就称为概率。也可用以下数学语言描述：

一般地，在大量重复试验中，如果事件 A 发生的频率 $\frac{m}{n}$ 会稳定在某个常数 p 附近，那么这个常数 p 就叫做事件 A 的概率，记作：

$$P(A)=p$$

根据前面的试验可看出，如果一件事情发生的概率是 $\frac{1}{n}$，不是指 n 次事件里必有一次发生该事件，而是指此事件发生的频率接近于 $\frac{1}{n}$ 这个数值。

前面介绍的两个例子中，足球比赛中的猜硬币正面或围棋比赛中猜棋子的单双，都只有两种情况发生，要么是硬币的正面，要么是反面（或者棋子数要么为单数，要么为双数），也就是只有两个基本事件。因此，每一种基本事件（如出现硬币正面）发生的概率都为 $\frac{1}{2}$，这个比较好理解。可表示为如图 6-5 所示的形式。





P(“正面朝上”)= $\frac{1}{2}$　　P(“反面朝上”)= $\frac{1}{2}$

图 6-5

下面，我们看一个相对复杂一点的例子：掷骰子。

一粒骰子有 6 个面，每面有不同的点数（分别为 1～6 点），那么，1～6 点出现的概率分别是多少呢？

在掷骰子时，骰子的 6 面都有相同的机会出现在最上面。如图 6-6 所示，骰子的每一面都表示一个基本事件，并且每个基本事件出现的可能性是相等的，因此，1～6 点出现的概率是相同的，其值为：

$$P(\text{“1点”})=P(\text{“2点”})=P(\text{“3点”})$$
$$=P(\text{“4点”})=P(\text{“5点”})=P(\text{“6点”})$$
$$=\frac{1}{6}$$

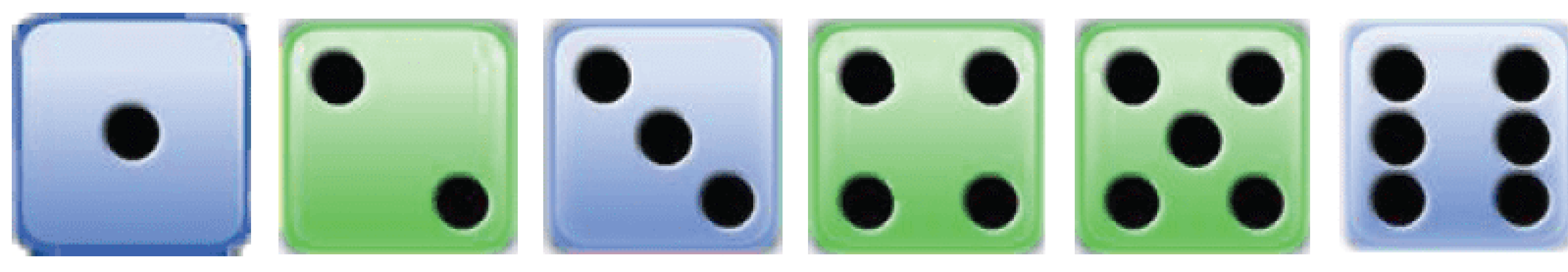


图 6-6

6.1.4　必然事件与不可能事件

在上面的例子中，抛掷硬币时，有硬币“正面向上”和“正面向下”两个事件；抛掷骰子时，有 1 点至 6 点这 6 种事件。也就是说，针对某种活动，可能会有多种事件之一发生，为了更好地分析概率，我们将这些事件分为 3 种类型，分别是：必然事件、不可能事件、随机事件。其中，必然事件和不可能事件又叫确定事件，而随机事件则是不确定事件。

1．必然事件

在一定的条件下重复进行试验时，有的事件在每次试验中必然会发生，这样的事件叫必然发生的事件，简称必然事件。

例如，太阳从东方升起，就是一个必然事件。再如，将 3 件不合格产品混在一堆合格产品中，再从这些产品中任意抽取 4 件，那么，其中必有 1 件是正品。显然，这也是一个必然事件。

可以看出，必然事件发生的概率肯定为 1。不过，概率为 1 的事件不一定为必然事件。

2．不可能事件

在一定的条件下重复进行试验时，不可能发生的事件叫不可能事件。

例如，太阳从西边升起，就是一个不可能事件。再如，春天过后是冬天，也是一个不可能事件。

可以看出，不可能事件的概率为 0。不过，概率为 0 的事件不一定为不可能事件。

3．随机事件

前面说过，必然事件和不可能事件都属于确定事件，即能确定地知道事件必然发生或必然不发生。与此对应，还有很多事件是处于两者之间，即在一定条件下，可能会发

生，也可能不发生的事件，这类事件称为随机事件。

例如，抛掷硬币时，“正面向上”的事件有可能发生，也可能不发生。类似地，抛掷骰子时，出现“1 点”的事件有可能发生，也可能不发生。

可以看出，随机事件的概率介于 0 与 1 之间。

下列事件中哪些事件必然发生？哪些事件必然不会发生？哪些事件可能会也可能不会发生？

（1）我今天买的彩票，中 500 万元大奖。（[redacted]）

（2）明天市内将下大雨。（[redacted]）

（3）边长为 3cm、4cm、5cm 的三角形是直角三角形。（[redacted]）

（4）某射击运动员一枪命中靶心。（[redacted]）

（5）肥皂泡会破碎。（[redacted]）

（6）股市上证指数今天将从昨天的 2200 点上涨到 6000 点。（[redacted]）

上面的 6 个事件中，前 5 个事件都比较好理解，最后一个事件是股票市场的，根据我国股票市场交易制度中的涨跌幅限制，即使所有股票今天全部涨停，上证指数也不可能从 2200 点上涨到 6000 点，因此，这是一个不可能发生的事件。

6.1.5 概率的基本性质

对概率有一定了解后，我们来总结一下概率的基本性质。

1. 概率P(A)的取值范围

- ❑ 0≤P(A)≤1；
- ❑ 必然事件的概率为 1；
- ❑ 不可能事件的概率为 0。

2. 任何两个基本事件是互斥的

针对某一个实验，可能有多个基本事件，但任何两个基本事件都是互斥的。这是比较好理解的。

例如，在抛掷硬币这个试验中，有两个基本事件：正面朝上、反面朝上，不可能出现既正面朝上又反面朝上的情况。

同样，在猜围棋子单双的实验中，也有两个基本事件：棋子数量为单数、棋子数量为双数，不可能出现棋子数量既为单数又为双数的情况。

在掷骰子的实验中，如果掷一粒骰子，有 6 个基本事件：分别为 1 点、2 点、3 点、4 点、5 点、6 点。这 6 种基本事件也是互斥的，不可能同时出现 1 点和 2 点。

3. 任何事件都可以表示成基本事件之和

先看一个例子：在抛掷骰子时，出现偶数点的概率为多少？

我们知道，骰子共有 1～6 点，其中 1、3、5 为奇数点，2、4、6 为偶数点，因此，

可以很容易地得出：

$$P(\text{“偶数点”})=\frac{1}{2}$$

即，出现偶数点的概率为$\frac{1}{2}$。

接下来，我们看看出现“偶数点”这个事件与骰子的几个基本事件之间的关系。

首先，抛掷骰子共有 6 个基本事件，分别是：P（“1 点”）、P（“2 点”）、P（“3 点”）、P（“4 点”）、P（“5 点”）、P（“6 点”）。

可以看出，“偶数点”这个事件与 3 个基本事件相关，即事件 P（“偶数点”）包含了 3 个基本事件，分别是：P（“2 点”）、P（“4 点”）、P（“6 点”）。

$$\begin{aligned}P(\text{“偶数点”})&=P(\text{“2点”})+P(\text{“4点”})+P(\text{“6点”})\\&=\frac{1}{6}+\frac{1}{6}+\frac{1}{6}\\&=\frac{1}{2}\end{aligned}$$

即，事件“偶数点”等于构成该事件的 3 个基本事件的概率之和。

4．事件的包含关系对概率的影响

若两个事件 A 和 B 之间存在包含关系，如 A⊆B，则事件 A 的概率将小于等于事件 B 的概率，即：

$$P(A)\leqslant P(B)$$

这个结论应该很好理解。前面掷骰子的例子中，“2 点”这个事件就包含于“偶数点”这个事件中，因此，其概率也小于“偶数点”的概率，即：

$$P(\text{“2点”})\leqslant P(\text{“偶数点”})$$

6.2　百枚钱币鼓士气

在现代数学中，概率论是非常有用的，这门学科在现代生产、生活及军事等各个领域中都有广泛的应用。下面我们来看一个用“概率”鼓舞士兵士气的故事。故事的主人公叫狄青，是北宋仁宗时期有名的大将。

最初，狄青只是宋朝防守陕西保安（现志丹县）的一名士兵。当时，西夏多次打败宋朝的军队，后来，狄青主动要求担任先锋出战。他披头散发，带上一个狰狞的面具，

带头冲入敌阵，把敌人打败。由于狄青屡立战功，被提升为将军。

后来，范仲俺召见了狄青，勉励他认真读书。从此狄青刻苦读书，精研兵法，之后打仗更有勇有谋，终因战功显赫被提升为掌管全国军事的枢密使。

6.2.1 狄青的计谋

在宋仁宗初年，我国南方少数民族的领袖侬智高自立政权，进攻现在的广西一带地区，占领了大片土地，打了不少胜仗，北宋朝野震动。为了平息叛乱，宋仁宗派狄青作为大将前去征讨侬智高。由于前期宋朝士兵与侬智高的士兵作战时基本上都是打败仗，宋朝士兵的士气非常低落。

狄青为了克服将士们的畏敌情绪，想出了一个办法。他立了一个神坛，当着全体将士的面向上苍祷告："如果这次上天保佑一定能打胜仗，那么，我把手中的一百枚铜钱扔到坛前地上时，钱面（铸文字的一面）一定全部朝上。"说完，在众目睽睽之下，他把 100 枚钱全部抛向地面，结果这 100 枚钱竟全部朝上，如图 6-7 所示。

于是全军欢呼震天动地。狄青命亲兵取来 100 枚大钉把钱全部钉在地上，任士兵观看，并说："待破敌凯旋，再来感谢神灵。"

将士们都认定肯定有神灵护佑，于是士气大振。所以在战斗中以一当百，奋勇无敌，果然连战皆捷，迅速平定了侬智高的叛乱。

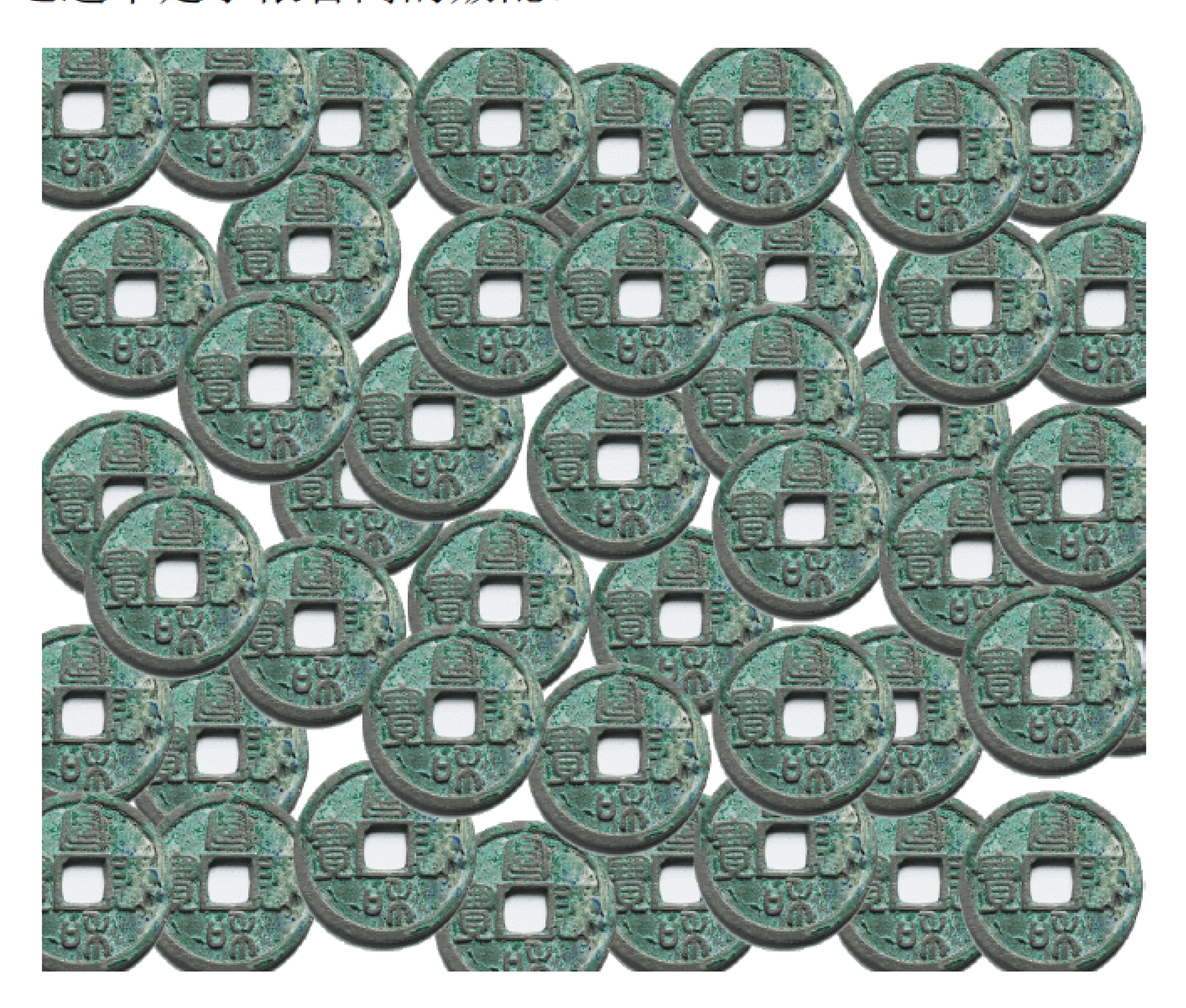


图 6-7

6.2.2 全为正面的概率是多少

为什么兵士们认为 100 枚铜钱的钱面全部朝上就一定受到神灵护佑呢？

这是因为，大家知道 100 枚铜钱全部朝上几乎是不可能的。下面我们来分析一下。

当我们抛下 1 枚铜钱时，有两种不同结果，钱面可能朝上，也可能朝下，如图 6-8 所示。




图 6-8

根据前面介绍的概率基础知识可以知道，当抛下 1 枚铜钱时，有两个基本事件：钱面朝上和钱面朝下。因此，钱面朝上的概率为：

$$P(\text{“钱面朝上”})=\frac{1}{2}$$

如果抛 2 枚铜钱，钱面朝上的概率为多少呢？先来看一下抛 2 枚铜钱有几个基本事件？如图 6-9 所示，抛掷 2 枚铜钱时共有 4 种基本事件。



图 6-9

假设铜钱有正、反两面（字面为正面，无字面为反面），从图 6-9 中可看出，4 枚铜钱共有以下 4 种基本事件：

- （正、正）；
- （正、反）；
- （反、正）；

- ❑（反、反）。

从 4 种基本事件中看出，两枚铜钱全部正面朝上的概率为：

$$P(\text{“2铜钱钱面朝上”})=\frac{1}{4}$$

另外，从上面列出的 4 个基本事件可看出，（正、反）和（反、正）这两个基本事件中，都是 1 枚铜钱钱面朝上、1 枚铜钱钱面朝下。因此，如果不对两枚铜钱进行顺序编号，就会将这两种基本事件看作为同一种基本事件了。换句话说，在列举基本事件时，必须对铜钱进行编号。

继续扩展，抛 3 枚铜钱时，可有以下 8 种基本事件：

- ❑（正、正、正）；
- ❑（正、正、反）；
- ❑（正、反、正）；
- ❑（正、反、反）；
- ❑（反、正、正）；
- ❑（反、正、反）；
- ❑（反、反、正）；
- ❑（反、反、反）；

在这 8 种基本事件中，3 枚铜钱全部钱面朝上的概率为：

$$P(\text{“3铜钱钱面朝上”})=\frac{1}{8}$$

可以看出，当抛 3 枚铜钱时就出现了 8 种基本事件，要列举出 3 枚铜钱正、反两面组合成的基本事件是比较麻烦的，并且很容易遗漏。

其实，对于作为程序员的我们来说，可以用更容易理解的方式来解决这个问题。可以将铜钱的正、反面看作二进制中的 0 和 1。多枚铜钱可按编号组合得到不同的基本事件，此时一枚铜钱就可看作为一位两进制位。

这样，当抛 1 枚铜钱时，就相当于是 1 位二进制，有两种可能，即 0 或 1（钱面朝上或钱面朝下），即有两种基本事件。

当抛两枚铜钱时，就相当于是两位二进制，有 4 种可能，分别是 00、01、10、11，即有 4 种基本事件。

当抛 3 枚铜钱时，就相当于是 3 位二进制，有 8 种可能，分别是 000、001、010、011、100、101、110、111，即有 8 种基本事件。

类似地，当抛 4 枚铜钱时，就相当于是 4 位二进制，有 16 种可能，即有 16 种基本事件。

依次类推，当抛 100 枚铜钱时，就相当于是 100 位二进制，有 2^{100} 种可能，即有 2^{100} 种基本事件。

求出基本事件数量后，再计算其概率就很简单了，只需要取其倒数即可。因此，当抛 100 枚铜钱时，钱面全部向上的概率为：

$$P(\text{“100铜钱钱面朝上”})=\frac{1}{2^{100}}$$

可以看出，这个概率值非常小，趋近于 0。

因此，100 枚铜钱钱面朝上几乎是不可能的事。而狄青抛掷 100 枚铜钱时，这种可能性微乎其微的事竟然发生了，将士们自然认为是有神灵护佑啰。

6.2.3　必然还是偶然

通过前面的计算，我们已经知道要使 100 枚铜钱钱面全部朝上的可能性微乎其微，其概率接近 0。虽然当时的兵士们还没有计算概率的数学知识，但根据大家日常生活中的基本认知，兵士们还是知道要使 100 枚铜钱钱面全部朝上的可能性非常小。

当然，狄青也知道这种可能性微乎其微，可他怎么会当着全体将士的面抛掷这 100 枚铜钱，并有信心使这些铜钱钱面全部朝上呢？

现在我们再来看一看，狄青带着部队凯旋回来的情况吧。当狄青命令把 100 枚钉子拔起时，他的下属将士们发现，原来，这些钱币都是狄青特制的，铜钱的两面都铸成了钱面！这样，不管铜钱两面中的哪一面朝上，都是钱面朝上。

也就是说，100 枚铜钱全部前面朝上是个必然事件！

在正常情况下，100 枚铜钱钱面全部朝上是一个偶然事件（只是 2^{100} 种基本事件中的一种）。但是，狄青通过特制铜钱，将这种偶然事件转换成了一种必然事件。

狄青只是利用了人们的思维定势，利用了人们敬畏鬼神的迷信心理，机智地采用偷梁换柱的手法，骗过了他的部下，鼓舞了士气，赢得了胜利。

6.3　庄家的胜率是多少

博彩业都是基于概率的，对一些看似公平的博彩游戏，通过仔细分析都会发现，庄家的胜率要大得多，否则的话，做庄家岂不是都要赔钱出局！下面我们再来看一个赌博中使用概率的例子。

6.3.1　一个看似公平的游戏

“碰运气”是在美国和海外很多赌场中玩的游戏。在游乐场中，操纵者为招来顾客而高声叫道：“每次三个人赢，三个人输！”这给人一个强烈印象，好像这个游戏是公平的。

那么，“碰运气”游戏是怎么玩的呢？

“碰运气”游戏是在一个笼子里装着三个骰子，翻转摇晃笼子就使骰子滚动，三个骰子将会出现三个不同的点数（也可能点数相同）。参与游戏的 6 名玩家可以赌三个骰子中出现 1～6 中的任何一个数，只要其中一个骰子出现玩家说的数时，他就赢了。如果有一个骰子与玩家说的数相同，玩家将赢得一份他赌的钱数；若有两个骰子与玩家说的数相同，玩家将赢得两份他赌的钱数；若三个骰子的点数都与玩家说的数相同，玩家将赢得三份他赌的钱数。如果玩家赌的数不是三个骰子中任何一个数，则玩家赌的钱将输给庄家。

例如，有 A、B、C、D、E、F 这 6 个人分别赌 1～6 这 6 个数（即 A 赌 1 点、B 赌 2 点、C 赌 3 点、D 赌 4 点、E 赌 5 点、F 赌 6 点），并分别赌一块钱。摇晃笼子后三个骰子分别出现如图 6-10 所示点数。

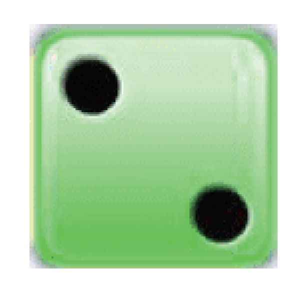
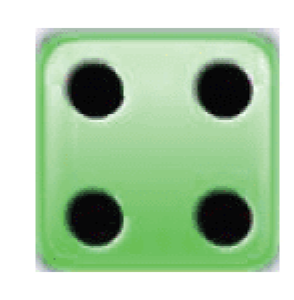




图 6-10

显然，玩家 A、C、F 这三个人赌的点数没有出现在图 6-10 所示的骰子点数中，因此这三个人将输掉自己所赌的一块钱，而玩家 B、D、E 这三个人赌的点数出现在图 6-10 所示的骰子点数中，因此，这三个人将各赢一块钱。也就是说，玩家 B、D、E 将赢得 A、C、F 这三个人所赌的钱。

看起来，这个游戏是很公平的，三个骰子出现点数的概率是相等的。

并且，玩家往往还会这样想：如果这个笼子里只有一个骰子，我赌的数就只能在 6 次中出现一次，概率为 $\frac{1}{6}$。如果有两个骰子，则 6 次中就会出现两次。有三个骰子时，6 次中就会有 3 次赢。这样看起来，玩家赢的机会还要大一些！如果赌一个数，例如赌 3 点，并赌一块钱，如果有一个骰子出现 3 点，则玩家可赢一块钱，如果有两个骰子是 3 点，就赢两块钱，如果三个骰子都是 3 点，就可赢三块钱!

怎么样？通常大家都会这样想吧？这也可以说明为什么赌场庄家会赢钱，会变成百万富翁！

6.3.2 庄家能赢钱吗

对于一个看起来很公平的游戏，有什么玄机让庄家能确保赢钱呢？让我们先来模拟一下游戏场景。

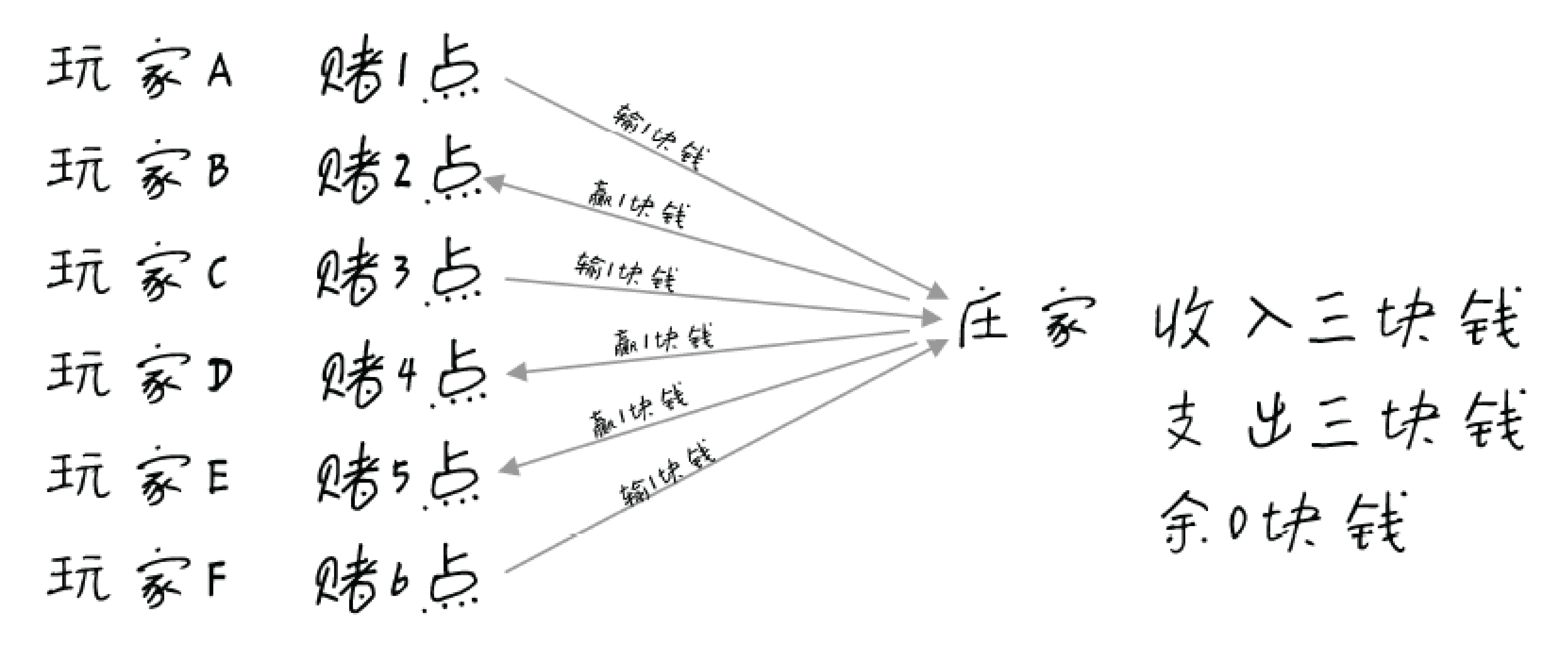
首先，6 位玩家分别赌 1～6 点，赌注为 1 块钱。

玩家A 赌1点

玩家B 赌2点

玩家C　赌3点
玩家D　赌4点　　赌注1块钱
玩家E　赌5点
玩家F　赌6点

接着摇晃笼子，得到 3 个骰子的点数如图 6-10 所示。玩家和庄家的输赢情况如下：



如果摇晃笼子，得到 3 个骰子的点数如图 6-11 所示。

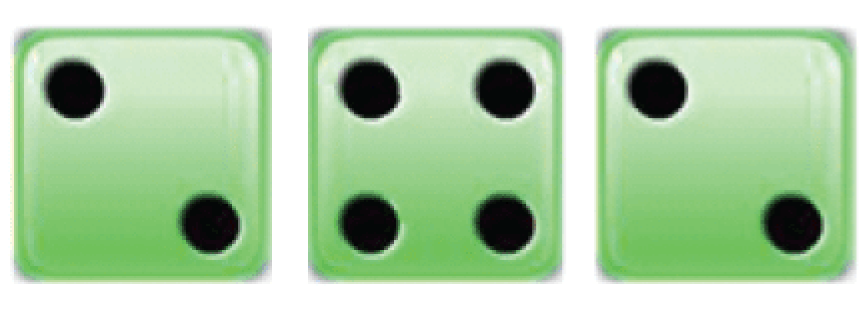
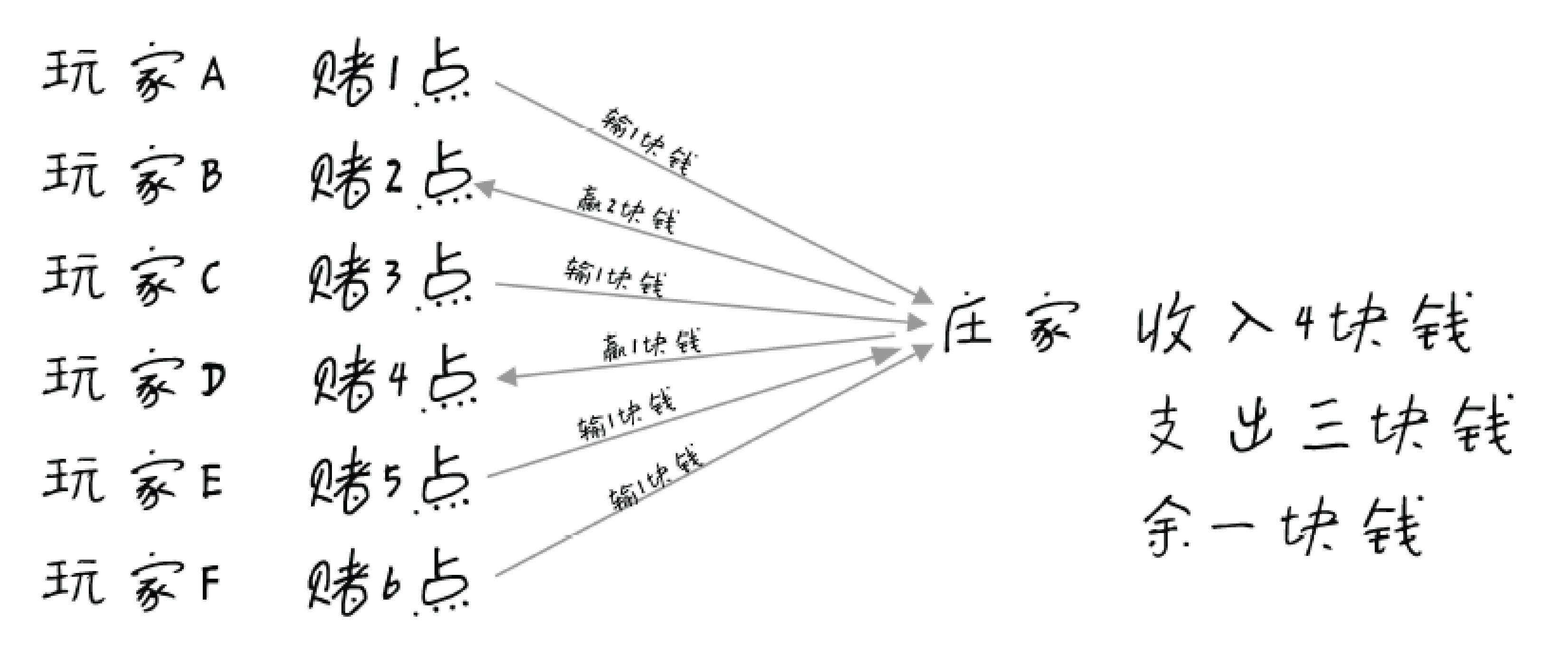


图 6-11

这时，只有两个玩家赌对了点数，4 个玩家赌错了点数，则玩家和庄家的输赢情况如下：



可以看出，当三个骰子中有两个骰子的点数相同时，庄家收入和支出之后将余一块钱，即庄家在这种情况下有了一块钱的收入。

如果摇晃笼子，得到三个骰子的点数如图 6-12 所示。

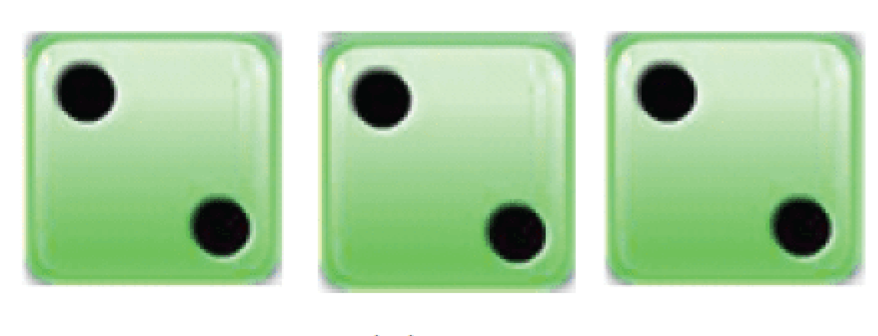
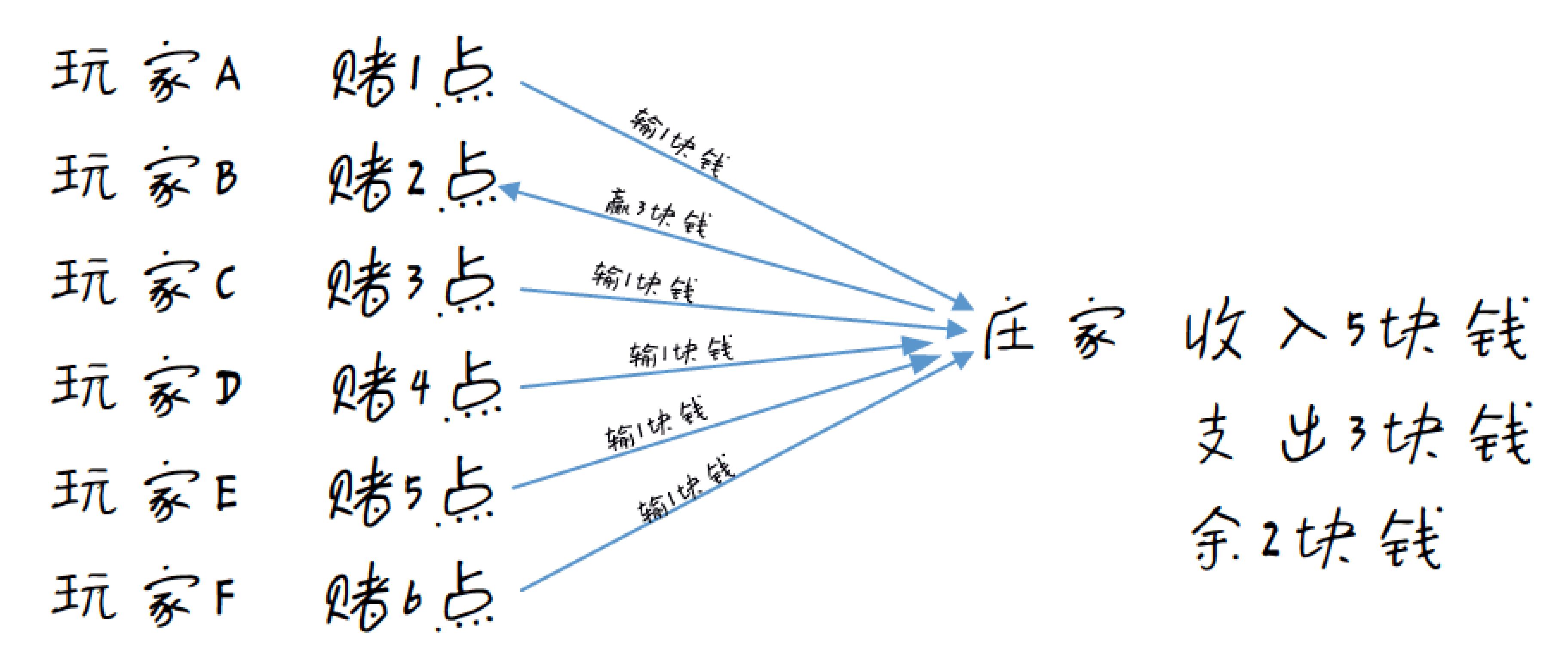


图 6-12

这时，只有一位玩家赌对了点数，另 5 位玩家都赌错了，则玩家和庄家的输赢情况如下：



可以看出，当 3 个骰子点数都相同时，庄家收入和支出相抵之后将余两块钱，即庄家在这种情况下有了两块钱的收入。

从以上的三种情况可看出，当笼子中的三个骰子点数各不相同时，庄家的收入为 0 块钱；当有两个骰子的点数相同时，庄家的收入为一块钱；当三个骰子的点数都相同时，庄家的收入为两块钱。

在整个过程中，庄家没有赔钱的情况，最坏的情况下也只是不赔不赚。因此，庄家在游乐场中开设这种游戏是稳赚不赔的！

6.3.3 庄家盈利比率

知道庄家玩这种“碰运气”游戏是稳赚不赔，那么庄家盈利比率是多少呢？

要计算庄家盈利比率，首先需要计算出基本事件及庄家获胜的事件，这样，就可方便地计算出庄家盈利的比率。

笼子中三个骰子共可组合出多少种基本事件呢？

每个骰子有 6 种基本事件，分别为 1～6 点。根据乘法原理，三个骰子共可组成

$$6 \times 6 \times 6=216$$

种基本事件。在这 216 种基本事件中，再分别计算三个骰子点数不同的基本事件、两个点数相同的基本事件、三个骰子点数相同的基本事件。

（1）三个骰子点数不同。可看作三个骰子分别取 1～6 这 6 个数字的排列，根据排列原理，三个骰子点数不同的基本事件数为：

$$P_6^3=6 \times 5 \times 4=120$$

（2）三个骰子中两个点数相同。可将三个骰子看作为两个骰子，这两个骰子分别取 1～6 中的两个数字的排列，可得到

$$P_6^2=6\times5=30$$

但是，这只是三个骰子中一个骰子与其他两个点数不同（其他两个骰子相同）的情况。类似图 6-13 所示，是第一个骰子点数与后两个点数不同的情况。

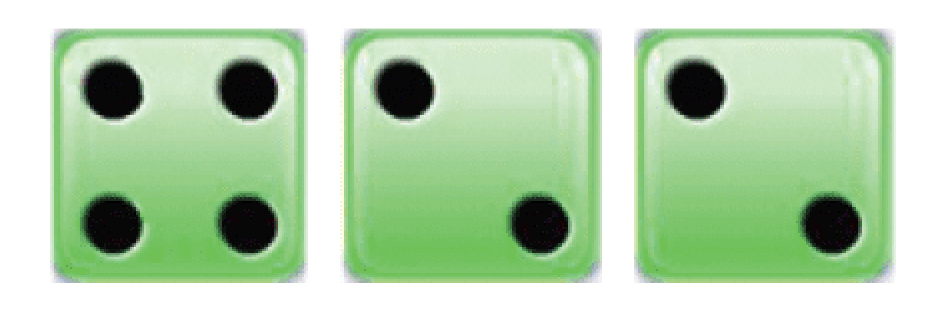


图 6-13

类似地，还有第两个骰子与其他两个点数不同的情况，以及第三个骰子与其他两个点数不同的情况。因此，共有三类，即三个骰子中两个点数相同的情况共有：

$$3\times30=90$$

在这 90 种基本事件中，庄家每次可赢 1 块钱。

（3）三个骰子点数相同。如果三个骰子点数相同，则相当于只有一个骰子，就只有 6 种基本事件了。

这三类基本事件相加可得到：

$$120+90+6=216$$

计算出各类基本事件之后，再计算庄家的获胜率就方便了。下面通过计算庄家收入和支出的钱数来计算庄家的胜率。

假设共举行 216 次游戏，则庄家在游戏开始前收入的钱为（赌注为一块钱，每次每位参与者付出一块钱赌注）：

$$216\times6=1296$$

当 3 个骰子点数不同时，庄家需要支付 6 块钱（返还三个赢家下注的三块钱，还要给三个赢家每人 1 块钱）。根据前面计算可知，三个骰子点数不同的基本事件共有 120 种，则庄家需要支付的钱为：

$$120\times6=720$$

当两个骰子点数相同时，只有两个玩家赢钱，庄家需要返还两个赢家下注的两块钱，同时赌中一个骰子点数的玩家赢一块钱，赌中两个骰子点数的玩家赢两块钱。因此，庄家需要支付 5 块钱。而这种两个骰子点数相同的基本事件有 90 种，则庄家需要支付的钱为：

$$90\times5=450$$

当三个骰子点数都相同时，只有一个玩家是赢家，庄家需要返还这个赢家下注的一

块钱，同时还要给赢家三块钱。因此，庄家需支付 4 块钱。而这种三个骰子点数相同的基本事件有 6 种，则庄家需支付的钱为：

$$6\times 4=24$$

这样，通过 216 次游戏，庄家收入 1296 块钱，支出 720+450+24 块钱，净赚的钱为：

$$1296-(720+450+24)=102$$

则，庄家盈利的比率为：

$$\frac{102}{1296}\times 100\%=7.87\%$$

也就是说，玩家每赌一块钱，庄家就会赚得 7.87 分，虽然看起来只有 7.87%的盈利。但是，由于庄家只赚不亏，因此，长期经营这个游戏，庄家将赚得不少。

6.3.4 游戏参与者获胜的概率

那么，参与“碰运气”游戏的玩家获胜的概率有多少呢？

我们在前面说过，如果笼子中只有一个骰子，玩家赌一个数时获胜的概率为$\frac{1}{6}$，如果笼子中有两个骰子，玩家获胜的概率为：

$$\frac{1}{6}+\frac{1}{6}=\frac{1}{3}$$

如果笼子中有三个骰子时，玩家获胜的概率为：

$$\frac{1}{6}+\frac{1}{6}+\frac{1}{6}=\frac{1}{2}$$

也就是说，玩家有一半的概率赢！

那么，实际情况真是这样吗？

下面，我们来计算一下实际情况。当玩家选择赌某一个数时，有三种赢的情况：

第 1 种，三个骰子点数相同，且是玩家所选的数。此时只有一种可能。

第 2 种，三个中有两个骰子的点数相同，且是玩家所选的数，此时另一个骰子的点数为其他 5 个数中任何一个，则玩家有 5 种可能获胜的情况。但是要注意，三个骰子是有位置顺序的，因此将单独数的那个骰子按顺序轮流，又会有 3 种变化。所以，玩家赌中两个骰子点数相同的情况有 3×5=15 种可能。

第 3 种，3 个骰子的点数各不相同，则只有一个骰子是玩家赌的数，此时其他两个

骰子可以是其余 5 个数中的任何一个，为 5×5=25 种可能。同样，三个骰子是有顺序的，三个骰子每个都可取玩家要的数，总共是 3×25=75 种。

因此，上述三种情况的可能性相加，得到：

$$75 + 15 + 1 = 91$$

也就是说，玩家在 216 种基本情况中有 91 种情况能获胜（即，只有 91 种情况是玩家赌的这个数至少出现一次），因此，玩家的胜率为：

$$\frac{91}{216} \times 100\% = 42.13\%$$

也就是说，玩家赢的机会并不是一半，而是比一半要小。

另外，需要注意，这里玩家获胜的概率与前面介绍的庄家盈利的比率是有区别的。庄家盈利是一个必然事件，只是盈利比率多少而已（即庄家是不会输的）。而玩家参与这个游戏是有输有赢的，获胜的概率表示玩家只有 41.55%的机率获胜，而（1–41.55%=）58.45%的机率会输。也就是说，实际上玩家赢的机率小于输的机率。

由此可见，在“碰运气”这个游戏中，庄家是稳赚不赔，而玩家则有赢也有输，且玩家输的机会更大，赢的机会更小。

6.4　你能中奖吗

彩票市场产生于 16 世纪的意大利，发展到今天，世界上已经有上百个国家和地区发行彩票，彩票业已成为世界第六大产业。

6.4.1　想中大奖吗

2011 年，我国彩票销售规模首次突破了 2000 亿元，达到 2215 亿元，彩票公益金筹集量达 634 亿元。

发行彩票集资可以说是现代彩票的共同目的。各国、各地区的集资目的多种多样，社会福利、公共卫生、教育、体育、文化是主要目标。以合法形式、公平原则，重新分配社会的闲散资金，协调社会的矛盾和关系，使彩票具有了一种特殊的地位和价值。

虽说彩票是以重新分配社会闲散资金为目的，不过，对于购买彩票者来说，总是冲着奖金去的，都想中奖，想中大奖。根据我国相关法律法规，单独彩票最高奖金为 500 万元（福利彩票“双色球”奖金最高可达 1000 万元）。也就是说，中了大奖就有可能获得高达 1000 万元的奖金，就可改善自己及家人的生活。

对于大多数彩民来说，都认为购买的彩票是否中奖属于自己的运气问题。但实际上

这是概率问题。下面，我们就来计算一下彩票中奖的概率。

要计算中奖概率，首先要计算出由符合要求的彩票号码组成的彩票数量，即将每一注彩票号码看作为一个基本事件，这样，就知道了所有基本事件。接着，计算根据彩票中奖规则能中奖的彩票号码数量。然后，通过以下公式就可计算出中奖概率：

$$中奖概率=\frac{中奖号码数量}{全部彩票号码数量}\times 100\%$$

现在，我国发行彩票的机构主要有民政局的福彩中心和国家体育总局的体彩中心。这两家机构发行的彩票种类又有很多种，下面我们以福彩中心发行的“双色球”为例，来计算这种彩票的中奖概率。

要计算彩票中奖的概率，首先必须了解该彩票的基本规则和中奖规则。

福利彩票“双色球”的基本规则是：有两种颜色的彩球，分别为红色和蓝色球，因此称为“双色球”。红球的号码从 1～33，蓝球的号码从 1～16。其中，每一注彩票号码由 7 个彩球的号码组成：从红球号码中选择 6 个号码，从蓝球号码中选择一个号码。

福利彩票“双色球”的中奖规则：分 1～6 共 6 个等级的奖项。

- ❑ 一等奖：一注彩票的 7 个号码全部相符（6 个红色球号码和一个蓝色球号码）（红色球号码顺序不限，下同），中奖者获取的奖金数据是浮动的，最高为 1000 万元。
- ❑ 二等奖：6 个红色球号码相符，中奖者获取的奖金数据是浮动的。
- ❑ 三等奖：5 个红色球号码和一个蓝色球号码相符，奖金 3000 元。
- ❑ 四等奖：5 个红色球号码或 4 个红色球号码和一个蓝色球号码相符，奖金 200 元。
- ❑ 五等奖：4 个红色球号码或 3 个红色球号码和一个蓝色球号码相符，奖金 10 元。
- ❑ 六等奖：一个蓝色球号码相符（有无红色球号码相符均可），奖金 5 元。

6.4.2 计算中奖概率

对福利彩票“双色球”的相关规则了解后，接下来就可计算出各奖项的中奖概率了。

1. 一等奖中奖概率

根据规则可知道，一等奖必须是 6 个红球号码和一个蓝球号码都与开奖号码相符，才算中了一等奖。也就是说，7 个号码要全部相符才能中奖，即所有彩票中只能有一注号码能中一等奖。

要计算一等奖的中奖概率，则需要将全部彩票号码列举出来，计算出共有多少注不同号码的彩票，即计算出彩票的基本事件。

根据福利彩票“双色球”的规则，可用第 4 章介绍的组合方法计算不同彩票号码的数量。首先，计算出 6 个红球可能的事件数量（从 1～33 个红球中选出 6 个红球）：

$$C_{33}^{6}=\frac{P_{33}^{6}}{P_{6}^{6}}$$

$$=\frac{33\times32\times31\times30\times29\times28}{6\times5\times4\times3\times2\times1}$$

$$=\frac{797448960}{720}$$

$$=1107568$$

也就是说，6 个红球可能会有 1107568 种不同的组合。

注意，还有一个蓝球，并且这个蓝球有 1～16 种号码可选择，因此还需要将 6 个红球的不同组合乘以 16，

$$1107568\times16=17721088$$

也就是说，“双色球”全部号码共有 17721088 种不同的组合，在这些组合中，只有一注能中一等奖。所以，一等奖中奖概率为：

$$P(\text{“一等奖”})=\frac{1}{17721088}\times100\%$$

$$=0.000005643\%$$

可以看出，要中一等奖的概率是多么低——近两千万分之一。按照概率，如果一直买一注相同的彩票，按每周开奖 3 次，一年 52 周，则中一等奖的时间最长可能需要：

$$17721088\div3\div52\approx113596$$

即按照概率分析的话，如果一直购买一注相同的彩票，那么最长需要 10 万余年才能中到一等奖。

2．二等奖中奖概率

计算出了一等奖中奖概率，二等奖的概率就很好计算了。根据规则，二等奖中需要 6 个蓝色球号码相符，而不管蓝色球的号码为多少。也就是说，在开出的号码中，买的 16 个蓝色球不管是什么号码都能中二等奖（应排除中一等奖的那一注），即在 17721088 注彩票号码中有 15 注二等奖。因此，二等奖中奖概率的计算公式如下：

$$P(\text{“二等奖”})=\frac{C_{16}^{1}-1}{C_{33}^{6}C_{16}^{1}}\times 100\%$$

$$=\frac{15}{17721088}\times 100\%$$

$$=0.00008464\%$$

也就是说，二等奖中奖概率约为百万分之一。

3. 三等奖中奖概率

同样地，要计算三等奖的中奖概率，需要先计算出在全部彩票号码中包含三等奖的注数。可是，这个数量的计算比较复杂。按规则，三等奖是中了 5 个红色球号码和 1 个蓝色球号码。例如，某一期“双色球”开奖号码如下（前 6 个数是红色球的号码，最后 1 个数是蓝色球号码）：

01　04　13　20　27　28　　03

对于这个已开出的号码，如果 7 个号码全中，则为一等奖。而 5 个红色球号码相同，就中三等奖，例如，以下号码都是三等奖：

(02)　04　13　20　27　28　　03

01　(03)　13　20　27　28　　03

01　04　(05)　20　27　28　　03

01　04　13　(14)　27　28　　03

01　04　13　20　(21)　28　　03

01　04　13　20　27　(29)　　03

…　…　　　　…　…

可以看出，在以上号码列表中圈出的红球号码与开奖的号码不相符，但其他 5 个红球号码与开奖号码相同。

根据以上分析可看出，从 6 个红球号码中选择 5 个号码就可组成一个三等奖号码(不管蓝球号码)，从 n 个数中挑选 m 个数，就是一个组合 C_6^5 。

另外还要注意的是，在上面的中了三等奖的号码列表中，圈中的数字可以是已开出的 6 个红球之外的任意一个号码，也就是说可从（33–6=27）个红球号码中任选 1 个，

也是一个组合 C_{27}^{1}。

即在开出的中奖号码中，能中三等奖的号码数量共有：

$$C_6^5 C_{27}^1 = 6 \times 27=162$$

也就是说，在“双色球”中共有 162 个号码可中三等奖。那么，三等奖的中奖概率为：

$$P(\text{“三等奖”})=\frac{162}{17721088} \times 100\%$$

$$=0.0009142\%$$

即三等奖的中奖概率约为十万分之一左右。

4. 四等奖中奖概率

接下来计算四等奖的中奖概率。根据规则，四等奖有两种情况：

- 5 个红色球号码与开奖号码相符。
- 4 个红色球号码和一个蓝色球号码与开奖号码相符。

那么，计算四等奖的概率需要分两种情况来计算。

首先来看第一种情况，5 个红色球号码与开奖号码相符的情况共有多少注。根据前面计算三等奖和二等奖的分析，可得：

$$C_6^5 C_{27}^1 (C_{16}^1-1) = 6 \times 27 \times 15=2430$$

再次说明，（$C_{16}^{1}-1$）是因为如果蓝色球也相符的话，就是三等奖了，这里的减 1 就是将三等奖的情况减去。

接下来计算第二种情况，就是 4 个红色球号码和 1 个蓝色球号码与开奖号码相符的情况。满足这种情况的彩票号码的注数为：

$$C_6^4 C_{27}^2 =\frac{6 \times 5 \times 4 \times 3 \times 2}{4 \times 3 \times 2 \times 1} \times \frac{27 \times 26}{2 \times 1}$$

$$=5265$$

综合以上两种情况，则全部双色球号码中包含四等奖号码的数量为：

$$2430 + 5265 = 7695$$

那么，四等奖的中奖概率为：

$$P(\text{“四等奖”})=\frac{7695}{17721088} \times 100\%$$

$$=0.04342\%$$

即四等奖的中奖概率约为万分之四左右。

5. 五等奖中奖概率

接下来计算五等奖的中奖概率。根据规则，五等奖有两种情况：

- 4 个红色球号码与开奖号码相符。
- 3 个红色球号码和一个蓝色球号码与开奖号码相符。

那么，计算五等奖的概率也需要分两种情况来计算。

首先来看第一种情况，4 个红色球号码与开奖号码相符的情况共有多少注。根据前面计算四等奖的分析，可得：

$$C_6^4 C_{27}^2 (C_{16}^1 - 1) = 5265 \times 15 = 78975$$

再次说明，（$C_{16}^1 - 1$）是因为如果蓝色球也相符的话，就是四等奖了。

接下来计算第 2 种情况，就是三个红色球号码和一个蓝色球号码与开奖号码相符的情况。满足这种情况的彩票号码的注数为：

$$C_6^3 C_{27}^3 \quad \frac{6 \times 5 \times 4}{3 \times 2 \times 1} \times \frac{27 \times 26 \times 25}{3 \times 2 \times 1}$$

$$= 58500$$

综合以上两种情况，则全部双色球号码中包含五等奖号码的数量为：

$$78975 + 58500 = 137475$$

那么，五等奖的中奖概率为：

$$P(\text{“五等奖”}) = \frac{137475}{17721088} \times 100\%$$

$$= 0.7758\%$$

即五等奖的中奖概率约为千分之八左右。

6. 六等奖中奖概率

最后计算六等奖的中奖概率。根据规则，一个蓝色球号码与开奖号码相符即为六等奖，也就是说，只要中了蓝色球号码，不管红球号码为多少，都为六等奖。当然，需要减去一、二、三、四、五等奖中已包含的蓝色球中奖号码的数量，则在全部双色球号码中六等奖号码的数量为：

$$C_{33}^6 - 1 - 162 - 5265 - 58500$$

$$= 1107568 - 1 - 162 - 5265 - 58500$$

$$= 1043640$$

则六等奖的中奖概率为：

$$P(\text{“六等奖”})=\frac{1043640}{17721088}\times 100\%$$

$$=5.8893\%$$

即六等奖的中奖概率约为百分之六左右。也就是说买 100 次，有 6 次左右的机会中六等奖。

7. 全中奖概率

将各奖项的中奖号码数量计算出来之后，就可计算出整个“双色球”的中奖概率。首先，将各奖项中奖的数量计算出来：

$$1+15+162+7695+137475+1043640$$

$$=1188988$$

也就是说，在 1770 多万注不同号码中，有 110 多万注号码可中奖。因此，中奖号码的概率为：

$$P(\text{“全部奖项”})=\frac{1188988}{17721088}\times 100\%$$

$$=6.7095\%$$

即购买“双色球”彩票的中奖概率小于 7%。

6.5 渔塘中有多少条鱼

除了前面介绍的彩票业之外，概率在日常生活中的应用也十分广泛。在本章最后一节中，我们再来看一些常用的应用案例。

6.5.1 该怎么估算渔塘中的鱼

老李在农贸市场开了一个鲜鱼销售店，经常要到乡村去整塘购买渔民养的鱼，并每天从池塘中打鱼到门店中进行销售。

在购买整塘的鱼时，需要根据鱼的重量与渔民签订合同，计算合同金额。可是，该怎么估算鱼塘中鱼的重量呢？

在估算时，应使鱼的损失尽量小，并能尽量准确地估算出鱼的重量，并且这种方法要让买卖双方都接受。

通常能想到的办法有以下几种。

第一种方法：首先想到的方法就是称重。将池塘中的鱼全部打捞上来，再称重，这样可以得到比较精确的重量。可是，将鱼全部打捞上来称重后再放回池塘，会造成很大一部分鱼会死掉。

第二种方法：可考虑一种将问题规模缩小的方法。即将鱼塘划分成相等的多个区域，然后将一个区域中的鱼打捞上来，进行称重。再将该区域中鱼的重量乘以划分的区域数量，即可得到总的重量。

可是这种方法的缺点也是很明显的。划分的区域通常不能达到均匀相等；另一方面，对划分的区域进行打捞时，由于鱼会四处游动，并不能保证可以将所划分区域中的鱼全部打捞上来。

由于这些缺点，导致最终计算出来的结果误差很大，从而使买卖双方不容易认可、接受。

第三种方法：使用概率来估算。本节我们将详细介绍这种方法。

6.5.2 用概率来估算

用概率来估算鱼塘中鱼的重量，可以达到比较准确并能被买卖双方接受的结果。下面介绍具体的实现方法，可按以下步骤进行操作。

（1）从鱼塘中捞出 100 条鱼进行称重。

（2）将这 100 条鱼做上标记，然后放入池塘中。

（3）等一段时间后，在池塘的不同位置随机撒网打捞，将一网打捞上来的鱼进行称重，并记录第（2）步中做好标记的鱼的数量，以及未做标记的鱼的数量。

（4）多次重复第（3）步。

（5）将记录的数据进行计算，计算出做了标记的 100 条鱼被打捞上来的概率。

（6）根据得到的概率即可估算出整个池塘中鱼的数量和重量。

下面以一个实际操作来看看这种通过概率估算池塘中的鱼重量的方法。

（1）首先从池塘中打捞出 100 条鱼，称出这 100 条鱼的重量为 216 公斤。

（2）在这 100 条鱼的鱼尾涂上红色油漆，然后将这些鱼放入池塘中。

（3）一小时之后，在池塘的不同地方多次撒网打鱼，并记录每次打捞上来的鱼的数量，包括有标记和无标记的鱼的数量。10 次打捞后的结果如表 6-3 所示。

表 6-3

打捞次数	1	2	3	4	5	6	7	8	9	10
有标记	2	3	1	3	4	3	1	2	2	1
无标记	25	23	18	22	25	18	10	15	14	13

有了以上表格中的数据，接下来就可以进行估算了。首先估算鱼塘中鱼的数量，将 10 次打捞的鱼中有标记的鱼的数量相加，得到：

$$2+3+1+3+4+3+1+2+2+1=22$$

而这 10 次打捞中，总的鱼的数量为：

$$\underset{\text{有标记}}{22}+\underbrace{25+23+18+22+25+18+10+15+14+13}_{\text{无标记}}=205$$

根据有标记的鱼数量与打捞上来总的鱼数量之比可计算出一个概率：

$$\frac{22}{205}=0.1073$$

表示每 10 条鱼中约有 1 条鱼是做了标记的。

得到有标记鱼的概率后，可方便地估算出池塘中总的鱼的数量：

$$\frac{100}{0.1073}\approx 932$$

计算出鱼的总数量之后，接下来再来估算鱼的重量。100 条做了标记的鱼的重量为 216 公斤，则每条鱼的平均重量为 2.16 公斤。所以，池塘中鱼的重量估算为：

$$932\times 2.16\approx 2013$$

即池塘中鱼的重量约为 2013 公斤。

6.5.3　用概率方法求 π 值

圆周率，一般以 π 来表示，是一个在数学及物理学中普遍使用的数学常数。圆周率代表圆周长和直径的比值，约等于 3.141592654。圆周率是一个无理数，即是一个无限不循环小数。从有文字记载的历史开始，这个数就引起了人们的兴趣。圆周率最早是出于解决有关圆的计算问题。因此，求出其尽量准确的近似值，就是一个极其迫切的问题。几千年来作为数学家们的奋斗目标，古今中外一代又一代的数学家为此献出了自己的智慧和劳动。直到 19 世纪初，求圆周率的值仍然是数学中的头号难题。

我们知道，现在计算圆周率的方法有很多种，常见的如概率法、割圆法、公式法等。这里，我们介绍用概率法求圆周率的值。

概率法计算 π 值是以概率和统计理论方法为基础的一种计算方法。将所求解的问题同一定的概率模型相联系，用计算机实现统计模拟或抽样，以获得问题的近似解。

假设有一个半径为 1 的圆，如图 6-14 所示。我们知道圆的面积为 πr^2，现在半径 r 为 1，则圆的面积为 π。那么图中阴影部分（四分之一圆）的面积就等于 $\frac{\pi}{4}$。通过概率法计算出阴影部分的面积，也就得到了 $\frac{\pi}{4}$，将阴影部分面积乘以 4 即可得到近似的 π 值。

使用概率法计算圆周率的具体过程如下。

在图 6-14 所示图形中，右上角的正方形面积为 1（因为圆的半径为 1），阴影部分面积为 $\frac{\pi}{4}$。利用随机函数产生横坐标的值 x 和纵坐标的值 y（这两个值都应在 0～1 之间），

接着判断由这两个随机数构成的点是否位于四分之一圆的区域内（阴影部分），若该点位于阴影区域内则进行累加计数。

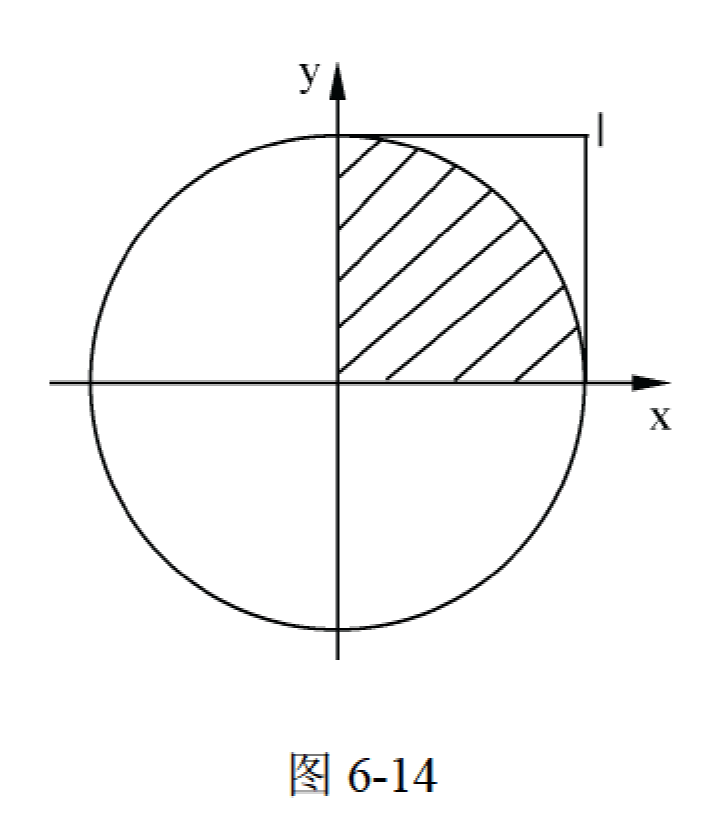


图 6-14

这样不断产生新的点，由于随机函数生成的点坐标有一定的均匀性，当生成的点足够多时，就可得到阴影内和阴影外点的近似均匀分布。

最后用统计的在阴影内的点的数量除以总的点数，即可得到近似的阴影面积，也就得到了一个 π 的四分之一近似值。

要判断产生的点是否位于阴影区域，可使用 $x*x+y*y<=1$ 进行判断。

按以上思路编写代码用来计算 π 的近似值，具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int i,n,sum=0;
    double x,y;

    printf("输入点的数量:");
    scanf("%d",&n);                    //输入产生随机点的数量

    srand(time(NULL));                 //初始化随机数生成器
    for(i=1;i<n;i++)
    {
        x=(double)rand()/RAND_MAX;     //产生 0~1 之间的一个随机数
        y=(double)rand()/RAND_MAX;     //产生 0~1 之间的一个随机数
        if((x*x+y*y)<=1)               //若在阴影区域
            sum++;                     //计数
    }

    printf("PI=%f\n",(double)4*sum/n); //输出结果
    getch();
    return 0;
}
```

编译执行以上程序，分别输入 10000 和 100000，可得到两个不同的结果，如图 6-15

所示。



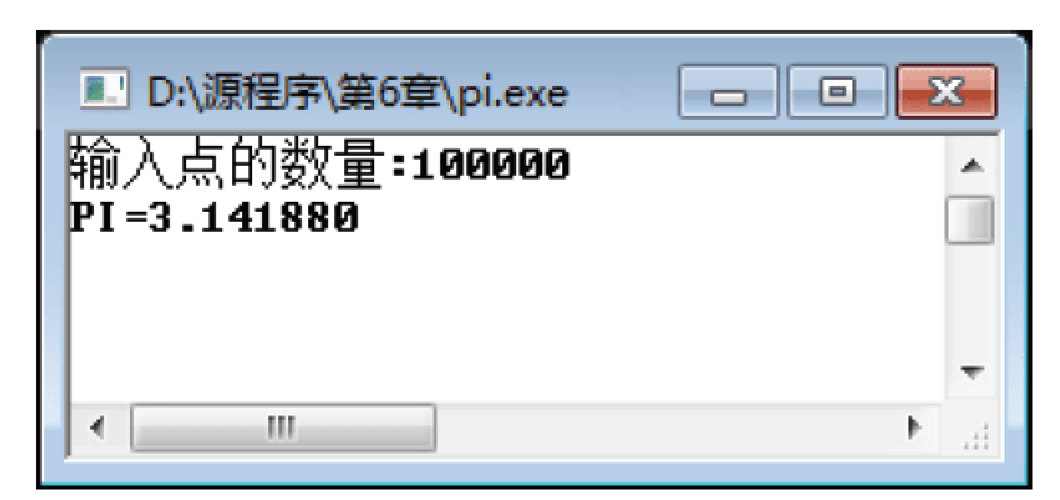


图 6-15

从图 6-15 中的结果可以看出，通过概率法计算的 π 值具有很大的随机性，在不同的运行时间，即使输入同样的点数，得到的结果也是不相同的，但其值会一直在 3.14 附近。

第 7 章　翻一番是多少

在现实生活中，经常听到“翻番”这个词语。那么，什么是翻番呢？我们这一章来研究这个数学问题，同时研究一下投资理财中常见的投资回报问题，包括单利和复利的相关内容。

7.1　翻番的概念

我们经常看到报道一个数据增加的概念：翻番。例如，在十八大中提出实现国内生产总值和城乡居民人均收入比 2010 年翻一番。这里的翻一番是什么意思？翻一番是多少？

首先来看一下翻番的概念，什么是翻番？另外还有一个翻倍的概念，这两者相同吗？

7.1.1　什么是翻番

从概念上来说，翻番就是数量加倍的意思，就是在原来的基础上加一倍，如果原来的基数是 100，翻一番之后就变成了 200。

那么，翻 2 番后是多少呢？翻一番很好理解，但翻 2 番就容易弄错了，通常我们会想成以下形式：

翻1番：　$100 \times (1+1)=200$

翻2番：　$100 \times (1+2)=300$　　（错误！）

以上计算方式是错误的！

翻 1 番是在基数的基础上进行一次翻番，翻 2 番则是需要进行 2 次翻番操作，即，进行如下所示翻番操作：

$$100 \xrightarrow[\times 2]{\text{翻1番}} 200 \xrightarrow[\times 2]{\text{再翻1番}} 400$$

也就是说，基数为 100，翻 2 番的结果如下：

$$100 \xrightarrow[\times 2 \times 2]{\text{翻2番}} 400$$

翻 *n* 番的结果可按以下方式计算：

$$100 \xrightarrow[\times 2 \times 2 \times \cdots \times 2]{\text{翻n番}} 100 \times 2^n$$

也就是说，翻 *n* 番就是用基数乘以 2 的 *n* 次方的结果。

例如，某企业现在年产值是 1000 万，要在 10 年内实现翻 2 番的目标。那么，该企业在 10 年后的产值应为：

$$1000 \times 2^2=4000$$

即 10 年后翻 2 番的产值为 4000 万元。

7.1.2　翻倍的概念

除了翻番这个概念外，我们还经常听到“翻倍”这个概念，这两者是相同的吗？

翻倍也是数据在基数的基础上增加的一种计量单位。翻 1 倍就是在基数的基础上增加 1 倍，翻 2 倍就是在基数的基础上增加 2 倍，翻 *n* 倍就是在基数的基础上增加 *n* 倍。

在大多数时候，我们都会将翻番和翻倍混同。例如，翻 1 倍和翻 1 番的计算结果就是相同的：

$$100 \xrightarrow[\times 2]{\text{翻1番}} 200$$

$$100 \xrightarrow[\times (1+1)]{\text{翻1倍}} 200$$

虽然计算结果相同，但是，注意其计算公式是不同的，也具有不同的含义。例如，翻 2 番和翻 2 倍的计算公式如下：

翻2番：　$100 \times 2^2=400$

翻2倍：　$100 \times (1+2)=300$

这就可以看出二者的区别了吧。如果基数是 100，翻 3 番的结果就是 800，而翻 3 倍的结果则为 400。

也就是说，翻 1 倍就是增加 100%；翻 1 番，也是增加 100%。除了 1 倍与 1 番相当外，2 倍与 2 番以上的数字含义就不同了，而且数字越大，差距越大。如增加 2 倍，是

指数据增加 200%；翻 2 番，则是 400%（1 番是 2，2 番是 4，3 番是 8），所以说翻 2 番就是增加了 300%，翻 3 番就是增加了 700%。“番”是按几何级数计算的，而“倍”则是按算术级数计算的。

7.1.3 计算倍数和番数

那么，如果知道目标数据和基数，怎么计算出增加的倍数和番数呢？

计算倍数的公式很简单，用目标数除以基数即可得到倍数，再减 1 即可得到增加的倍数，具体公式如下：

$$增加倍数=\frac{目标数}{基数}-1$$

例如，2012 年全国 GDP 为 519322 亿元，改革开放之初的 1978 年全国 GDP 为 3645 亿元，计算 2012 年相比 1978 年全国 GDP 增加的倍数为：

$$\frac{519322}{3645}-1\approx 141.5\ （倍）$$

而要计算翻了多少番的公式就比较复杂了，这时可以借助对数函数进行计算，具体公式如下：

$$番数=[\lg(目标数\div 基数)]\div \lg 2$$

例如，要计算 2012 年相比 1978 年全国 GDP 翻了几番，可用如下算式计算：

$$=[\lg(519322\div 3645)]\div \lg 2$$

$$\approx 2.15374\div 0.30103$$

$$\approx 7.1546$$

也就是说，2012 年全国 GDP 在 1978 年全国 GDP 基础上翻了 7 番多。

7.2 复利的威力

曾经有人这样问过爱因斯坦：“世界上什么威力最大？”爱因斯坦不假思索地答道：“复利，复利的威力比原子弹还可怕！”。

那么，什么是复利？复利有什么奥秘？

通过复利，在多长时间能使我们的投资翻番？

下面我们就来探索复利的问题，在这之前，先了解一下投资回报的概念。

7.2.1　利润——投资回报

随着国家 GDP 不断翻番增长，大家的收入也在不断增长。为了使自己的财富保值增值，投资理财也成为了大家很关心的话题。对于投资理财，大家最关心的就是投资回报率。

所谓投资回报率，是指正常年度利润或年均利润占投资总额的百分比。其计算公式如下：

$$ROI = \frac{\text{年利润}}{\text{投资总额}} \times 100\%$$

投资回报率能反映投资的综合盈利能力，且由于剔除了因投资额不同而导致的利润差异的不可比因素，因而具有横向可比性（如投资 10 万与投资 100 万年利润金额是不可比的，但若用投资回报率来比较，就具有可比性），有利于判断不同投资项目业绩的优劣；此外，投资回报率可以作为选择投资机会的依据，有利于优化资源配置。

投资回报率这个概念显得很高深，其实，简单地说，就是投入的钱可获取的收益百分比。例如，将钱存在银行后可获得利息收入，就是一种投资回报。而利率通常是按百分比来显示的，如一年期定期存款的年利率是 3%，表示将 100 元钱存入银行一年，可获取 3 元的利息。

当然，除了将钱存在银行获取利息收入这种投资方式之外，现在我们还可以有多种投资理财方式。例如，购买股票、基金、债券、国券等，或购买房产、投资实业公司等。对于这些投资项目，都可能给投资者带来收益。根据前面介绍的公式，将年利润除以投资总额即可得到该投资项目的投资回报率。

7.2.2　认识单利

通常，理财项目都会公布预期收益率（也就是投资回报率），对于这些收益率，投资者需要注意是单利模式还是复利模式。

所谓单利，是指仅对本金计息的利息计算方式。

单利方式计算投资的利润收入的公式如下：

$$\text{利润} = \text{投资额} \times \text{利率} \times \text{投资期}$$

通过单利方式投资后获取的利润和本金的总收入可用以下公式计算：

$$\text{总收入} = \text{投资额} \times (1 + \text{利率} \times \text{投资期})$$

了解这些计算公式后，下面来计算一个投资项目的收益。

老张准备为儿子存储 50000 元教育基金，5 年后才会支取使用。为了保障资金的安

全，老张准备将这笔钱在银行存定期。根据银行的定期存款期限，可以选择存 5 个 1 年期（即每年期满后再续存），也可以选择存 1 个 5 年期，还可选择存 1 个 2 年期和 1 个 3 年期。那么，在这些存款方式中，哪种方式可获取更高的收益？下面我们帮老张计算一下。

首先，通过网络查询得到如表 7-1 所示的 2012 年 7 月 6 日开始执行的存款利率表。

表 7-1 存款利率表

项 目	年利率（%）
一.城乡居民及单位存款	
（1）活期	
（2）定期	0.35
1.整存整取	
三个月	2.85
半年	3.05
一年	3.25
二年	3.75
三年	4.25
五年	4.75
2.零存整取、整存零取、存本取息	
一年	2.85
三年	2.90
五年	3.00
3.定活两便	按一年以内定期整存整取同档次利率打 6 折执行
二.协定存款	1.15
三.通知存款	
一天	0.80
七天	1.35

从表 7-1 中可看到，1 年期整存整取的年利率为 3.25%。如果将 50000 元存 5 次 1 年期，则总收入为：

$$50000 \times (1+3.25\% \times 5) = 58125$$

如果将 50000 元存 1 次 2 年期、1 次 3 年期（2 年期整存整取的年利率为 3.75%，3 年期整存整取的年利率为 4.25%），则总收入为：

$$50000 \times (1+3.75\% \times 2+4.25\% \times 3) = 60125$$

如果将 50000 元存 1 次 5 年期（5 年期整存整取的年利率为 4.75%），则总收入为：

$$50000 \times (1+4.75\% \times 5) = 61875$$

将以上计算结果汇总到如表 7-2 所示的表格中，在这张表中计算出了每一年的利息收入。从表中的结果可看出，一次性存 5 年整存整取的收益最高，本利合计为 61875 元，

与存 5 次 1 年期整存整取相比要多出 3750 元。

$$61875 - 58125 = 3750$$

表 7-2　有款利率汇总表

	1 年期定存	2+3 年期定存	5 年期定存
利率	3.25%	3.75% 4.25%	4.75%
第 1 年	1625	1875	2375
第 2 年	1625	1875	2375
第 3 年	1625	2125	2375
第 4 年	1625	2125	2375
第 5 年	1625	2125	2375
总收入	58125	60125	61875

可以看出，同样将钱存到银行，选择不同的存款期限获取利息的差距还是很大的，5 个 1 年期存款的利息收入与 1 个 5 年期存款的利息收入相差 3750 元，比例为 46.15%。

$$\frac{3750}{8125} \times 100\% \approx 46.15\%$$

7.2.3　认识复利

在表 7-2 所示的 3 种存款方式中，可以明显看出定存期限越长，银行支付的利率越高，最后获取的利息也就越多。

不过我们也发现一个问题，按 1 年期定存时，既然每年都要重新做一次 1 年期定存操作，为什么不可以将上一年度存款获取的利息也一起存呢？这样获得的利息肯定也更多。

如果读者已经意识到这个问题了，那么恭喜你找到了一条致富的途径：复利。

什么是复利呢？就是将上期利息加入本金，然后一并计算利息的一种方法。例如，在按 1 年期定存时，第 1 年期满后取得利息，第 2 年定存时将本金和利息一并存入。在第 2 年期满后获取的利息又与本金一起定存 1 年，如此周而复始，就是复利，民间俗称“利滚利”。

下面，我们就来计算 1 年期定存 50000 元，并按复利形式计息，5 年后本利合计共是多少钱。具体计算过程如表 7-3 所示。

表 7-3　1 年期定存（复利）

年利率：3.25%

	本　　金	利　　息	本 利 合 计
第 1 年	50000.00	1625.00	51625.00
第 2 年	51625.00	1677.81	53302.81
第 3 年	53302.81	1732.34	55035.15

年利率：3.25%

	本　　金	利　　息	本 利 合 计
第 4 年	55035.15	1788.64	56823.80
第 5 年	56823.80	1846.77	58670.57

从表 7-3 中可看出，在复利计算时，总是将前一期的本利合计作为下一期的本金。例如，第 1 期期满后的本利合计用如下公式计算：

$$第1期本利合计 = 本金 \times (1 + 利率)$$

第 2 期的本利合计用如下公式计算：

$$第2期本利合计 = 第1期本利合计 \times (1 + 利率)$$

而将“第 1 期本利合计”的公式代入上式的公式，可得：

$$\begin{aligned} 第2期本利合计 &= 本金 \times (1 + 利率) \times (1 + 利率) \\ &= 本金 \times (1 + 利率)^2 \end{aligned}$$

类似地，第 3 期的本利合计的公式如下：

$$第3期本利合计 = 本金 \times (1 + 利率)^3$$

第 n 期本利合计的公式如下：

$$第n期本利合计 = 本金 \times (1 + 利率)^n$$

在表 7-2 中列出的是单利方式得到的本利合计，对比表 7-2 和表 7-3 可看出，1 年期定存 5 年，如果采用单利方式，本息合计为 58125 元，而采用复利方式，本息合计为 58670 元。也就是说，复利方式比单利方式多获得利息 545 元，与单利方式相比，可多获取 6.7%的利息收入。看起来 5 年利息收入只多 6.7%，绝对值也只有 545 元，不算多。可是，如果将存钱的时间拉长，就可看出复利的威力了。如表 7-4 所示，是按 1 年期存款的单利和复利的对比（按 20 年为周期对比）。

表 7-4　1 年期定存

年利率：3.25%　本金：50000

期　　间	单　　利	复　　利	期　　间	单　　利	复　　利
1	51625.00	51625.00	11	67875.00	71082.17
2	53250.00	53302.81	12	69500.00	73392.34
3	54875.00	55035.15	13	71125.00	75777.59
4	56500.00	56823.80	14	72750.00	78240.36
5	58125.00	58670.57	15	74375.00	80783.17
6	59750.00	60577.36	16	76000.00	83408.63
7	61375.00	62546.13	17	77625.00	86119.41
8	63000.00	64578.88	18	79250.00	88918.29
9	64625.00	66677.69	19	80875.00	91808.13
10	66250.00	68844.72	20	82500.00	94791.90

从表 7-4 中可看出，经过 20 年，复利比单利多获得利息 12291.9 元，多 37.81%。也就是说，复利方式比单利方式获得的利息可多出三分之一。

$$94791.90 - 82500.00 = 12291.9$$

$$\frac{94791.9-82500}{82500-50000} \times 100\% = \frac{12291.9}{32500} \times 100\% = 37.81\%$$

根据表 7-4 中的数据可绘制出如图 7-1 所示的对比图，从图 7-1 中可看出，在单利方式下，利息收入与时间之间的关系呈现为一条直线，而复利方式下，利息收入与时间之间的关系呈现为一条向上弯曲的曲线。

上面的例子中都是以定期存款为例进行的计算。由于银行定期存款基本上无风险，因此其收益率也很低。如果做其他投资产品或投资实业，将获得更高的年收益率。假设投资某一项基金（或信托产品），年收益率为 10%。那么，按单利和复利方式计算，经过 20 年投资可获得多少收益？

具体计算过程如表 7-5 所示。

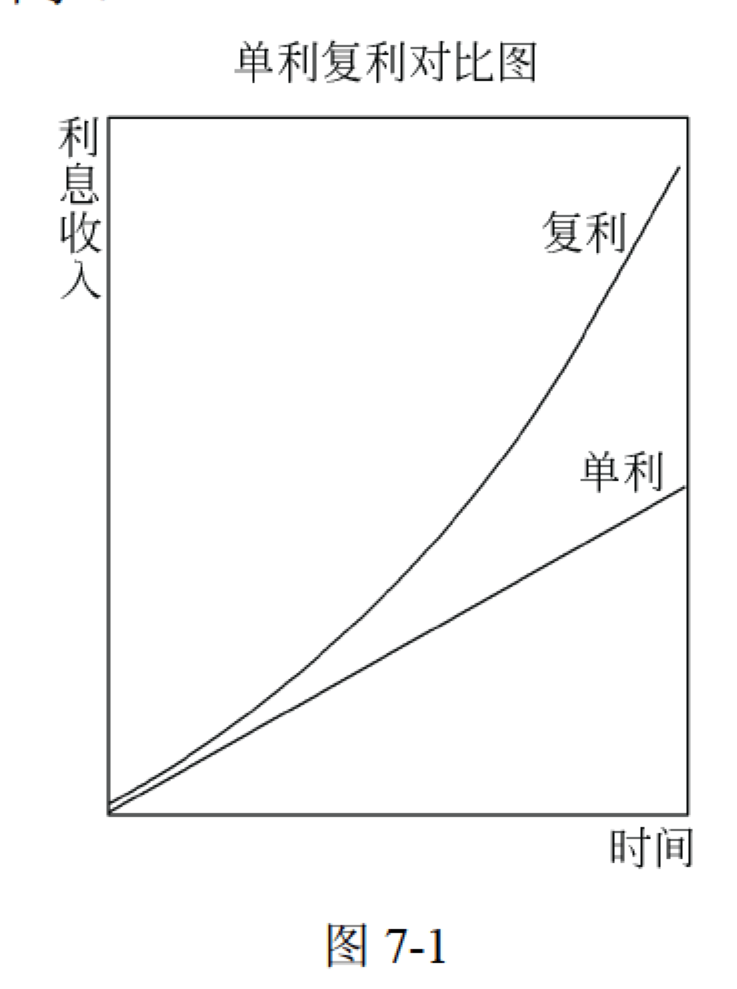


图 7-1

表 7-5　增加年收益后单利与复利的对比

年收益：10.00%　本金：50000.00

期　间	单　利	复　利	期　间	单　利	复　利
1	5000.00	5000.00	11	55000.00	92655.84
2	10000.00	10500.00	12	60000.00	106921.42
3	15000.00	16550.00	13	65000.00	122613.56
4	20000.00	23205.00	14	70000.00	139874.92
5	25000.00	30525.50	15	75000.00	158862.41
6	30000.00	38578.05	16	80000.00	179748.65
7	35000.00	47435.86	17	85000.00	202723.51
8	40000.00	57179.44	18	90000.00	227995.87
9	45000.00	67897.38	19	95000.00	255795.45
10	50000.00	79687.12	20	100000.00	286375.00

表 7-5 是按年收益率 10%，以 50000 元投资 20 年时，单利方式与复利方式的收益对比。从表 7-5 中可看出，单利方式的收益在第 10 年时可与最初投入的本金相等（即投资翻倍了）。而在复利方式下，第 7 年时的收益就接近翻一倍了。到第 20 年时，单利方式的收益是本金的 2 倍，而复利方式的收益是本金的 5-73 倍。

通过以上的分析可知道，复利的要素有 3 方面：

- 初始本金：初始投入的本金越大，复利获取的收益越多。
- 回报率：回报率越高，复利获取的收益越多。
- 时间：投资的时间越长，复利获取的收益越多。

7.2.4 计算投资回报的程序

通过前面演示的例子，作为程序员的我们，就可以编写程序来帮助我们快速计算出投资回报。在程序中，让用户输入初始投资额、回报率、投资期数等数据，然后就可快速计算出相应的利润、本利合计等数据。

具体的程序代码如下：

```
#include <stdio.h>
float fact(float f,int n);                //计算阶乘
float ROI(float a,float r,int p);         //计算回报

int main()                                //主函数
{
    float rate,amount,pay,t1;
    int period,i;

    printf("投资回报计算程序\n\n");
    printf("投资金额：");
    scanf("%f",&amount);

    printf("回报率：");
    scanf("%f",&rate);

    printf("投资期数：");
    scanf("%d",&period);

    printf("\n\n 期数\t 利润\t 本利合计\n");
    for(i=1;i<=period;i++)                //输出每一期的利润及本利合计
    {
        t1=ROI(amount,rate,i);
        printf("%d\t%.0f\t%.0f\n",i, t1-amount,t1);
    }

    getch();
    return 0;
}

float ROI(float a,float r,int p)          //计算投资回报
{
    return a*fact(1+r,p);
```

```
}

float fact(float f,int n)                //计算阶乘
{
    float temp=1;
    int i;
    for(i=0;i<n;i++)
    {
        temp*=f;
    }
    return temp;
}
```

由于 C 语言中未提供阶乘运算，因此，在以上程序中编写了 fact()函数用来计算阶乘；然后编写计算投资回报的函数 ROI。

运行以上程序，输入前面例子中计算过的数据，其中投资金额为 50000，回报资为 0.1（10%），投资期数为 20 期，则可得到如图 7-2 所示的运算结果。

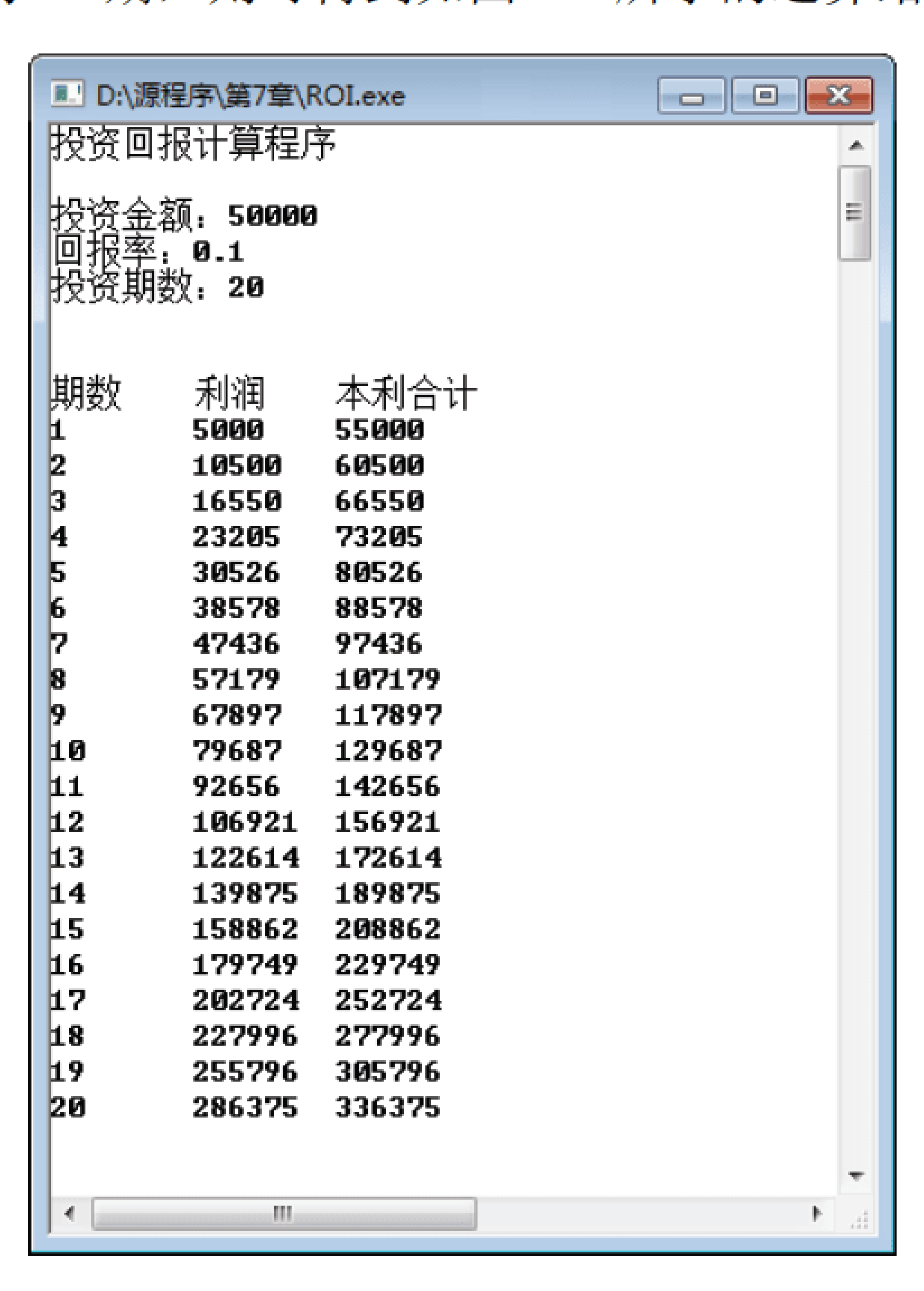


图 7-2

7.2.5　忘还钱的信用卡

信用卡是一种非现金交易的付款方式，是简单的信贷服务。持卡者只需要在商户的 POS 机中刷卡消费即可完成向银行的小额贷款服务。由于信用卡使用方便快捷，受到了很多用户（特别是年轻消费者）的欢迎。

可是，很多信用卡用户由于各种原因忘记了按时还款，由此导致出现了高额的利息。

可别忘了，信用卡的利息是按月进行复利计算的！那可是一笔高昂的费用！

目前我国绝大部分银行发行的信用卡透支利息计算方式是：上期对账单的每笔透支金额为计息本金，自该笔透支交易的记账日起至还款日为计息天数，按日利率万分之五计息，按月计收复利。另外每月还会按计息本金的 5%收滞纳金。

例如，最近媒体报道，有一个信用卡持卡人在 2008 年 10 月 3 日的欠款余额是 1057.86 元，此后，这张信用卡除了银行扣收滞纳金、利息、超限费，再没有发生任何一笔业务。结果到 2012 年底，银行的催收通知单显示，本息合计欠款 23000 多元。

欠款 1057.86 元，4 年多时间，怎么就变成 23000 元了呢？下面我们来帮这位持卡人计算一下，具体计算过程如表 7-6 所示。

表 7-6　信用卡利息计算

日利率：0.0005　月利率：0.015

月　　数	本　　金	利　　息	滞　纳　金	本　利　合　计
1	1057.86	15.87	52.89	1126.62
2	1126.62	16.90	56.33	1199.85
3	1199.85	18.00	59.99	1277.84
4	1277.84	19.17	63.89	1360.90
5	1360.90	20.41	68.05	1449.36
6	1449.36	21.74	72.47	1543.57
7	1543.57	23.15	77.18	1643.90
8	1643.90	24.66	82.20	1750.75
9	1750.75	26.26	87.54	1864.55
	……		……	
41	13134.47	197.02	656.72	13988.21
42	13988.21	209.82	699.41	14897.44
43	14897.44	223.46	744.87	15865.78
44	15865.78	237.99	793.29	16897.05
45	16897.05	253.46	844.85	17995.36
46	17995.36	269.93	899.77	19165.06
47	19165.06	287.48	958.25	20410.79
48	20410.79	306.16	1020.54	21737.49
49	21737.49	326.06	1086.87	23150.43

在表 7-6 的计算中，日利率为万分之五（0.05%），按每月 30 天计算，则月利率为 1.5%，转化为年利率达到 18%，这个利息不可谓不高，并且利息是按月进行复利计算的。也就是说，每个月账单中的本利合计作为下月的本金。当然，除了本金、利息之外，还有更大一部分费用就是滞纳金（为 5%），那可比利息更高。所以，通过近 50 个月的“利滚利”积累，该持卡人的还款额就从 1000 多元增涨到了 23000 多元了。

当然，以上算法只是按照银行规则进行的测算，实际上在不同银行的计算方法有些不同，最终的还款金额也会有所不同。

怎么样，再一次见识到复利的威力了吧！在信用卡利息的计算上，是按月计算复利，比我们前面例子中按年计算复利更厉害。

7.2.6　爱因斯坦的 72 法则

了解复利后，再利用爱因斯坦的 72 法则，我们就可以轻松算出自身价值何时翻倍。

爱因斯坦的 72 法则，也是金融学上有名的“72 法则”，是指 “用 72 除以增长率（回报率）”可快速估计出投资倍增或减半所需的时间，反映出的是复利的结果。

例如，如果初始本金为 10000 元，年回报率为 9%，则要使这笔投资本利合计翻倍（即达到 20000 元），可用以下算式快速计算出需要的时间：

$$72 \div 9 = 8$$

即表示经过连续 8 年这种固定收益的投资回报，就可使投资翻倍。是不是这样呢？下面我们进行逐年计算，得到如表 7-7 所示的结果。

表 7-7　72 法则验证

年回报率：9%

期　　数	本　　金	利　　息	本　利　合　计
1	10000.00	900.00	10900.00
2	10900.00	981.00	11881.00
3	11881.00	1069.29	12950.29
4	12950.29	1165.53	14115.82
5	14115.82	1270.42	15386.24
6	15386.24	1384.76	16771.00
7	16771.00	1509.39	18280.39
8	18280.39	1645.24	19925.63

从表 7-7 中可看出，经过 8 期的复利计算，最后本利合计为 19925 元，约为 20000 元。从这个表就可以验证出 72 法则是有效的。

通过上面的验证可知道，根据 72 法则就可快速估算出投资翻倍的期数。例如，1 年期定存每期（每年）的回报率（利率）为 3.25%，要使存款翻倍需要的期数（年数）为：

$$72 \div 3.25 = 22.15$$

也就是说，如果按 1 年期定存的方法进行投资，需要 22 年多点的时间才能让投资金额翻倍。而前面计算过，如果年回报率为 9%，则只需要 8 年时间就能让投资金额翻倍。如果投资回报率能达到更高的水平，则投资翻倍的时间将会更短。

其实，对于投资回报率，不要求有多高，只要稳定在一个合理的水平，经过较长的一个投资期，仍然可以得到可观的回报。例如，现在的华人首富李嘉诚，他从 16 岁开始创业一直到 85 岁时，白手起家 69 年，家产已达 310 亿美元。对于普通人来说，这是一个天文数字。

但是，我们仔细计算一下，如果我们有 10000 美元进行投资，每年的回报率能稳定达到 24.3%，用 69 年的时间，就可使投资的本利合计达到 330 亿美元。

在做股票之类的投资者看来，觉得一年 24.3%的收益率简直不值一提，经常会有年

收益率超过 100%的情况。但是，如果能长期稳定地保持在这一较低的回报水平（24.3%），经过长时间的投资，回报也是非常高的。

根据 72 法则来计算，24.3%的回报率，投资翻倍需要的年数为：

$$72 \div 24.3 = 2.96$$

也就是说，经过 3 年的投资，10000 美元的本金就可变成 20000 美元。按这个收益率，每过 3 年就可使投资额翻番，因此，经过 69 年，最初 10000 美元的本金翻 20 多番后就可达到一个天文数字。

7.3 对折纸张

我们已经知道翻番就是使得到的数为基数的 2 倍，下面我们再来研究与翻番相反的问题——折半，即指每次得到的数为基数的一半。

这节我们研究一个看似简单的问题——折纸，就是将纸张进行多次对折。这个操作可以比较直观地看到折半问题的效果。

7.3.1 有趣的问题：纸张对折

如图 7-3 所示，左边是一张纸，沿中线对折后得到右图的效果，这样就算完成了一次对折。接着看这样一个问题：一张纸能对折多少次？

这个问题看起来很简单，感觉应该能对折很多次，至少不会少于 10 次吧。

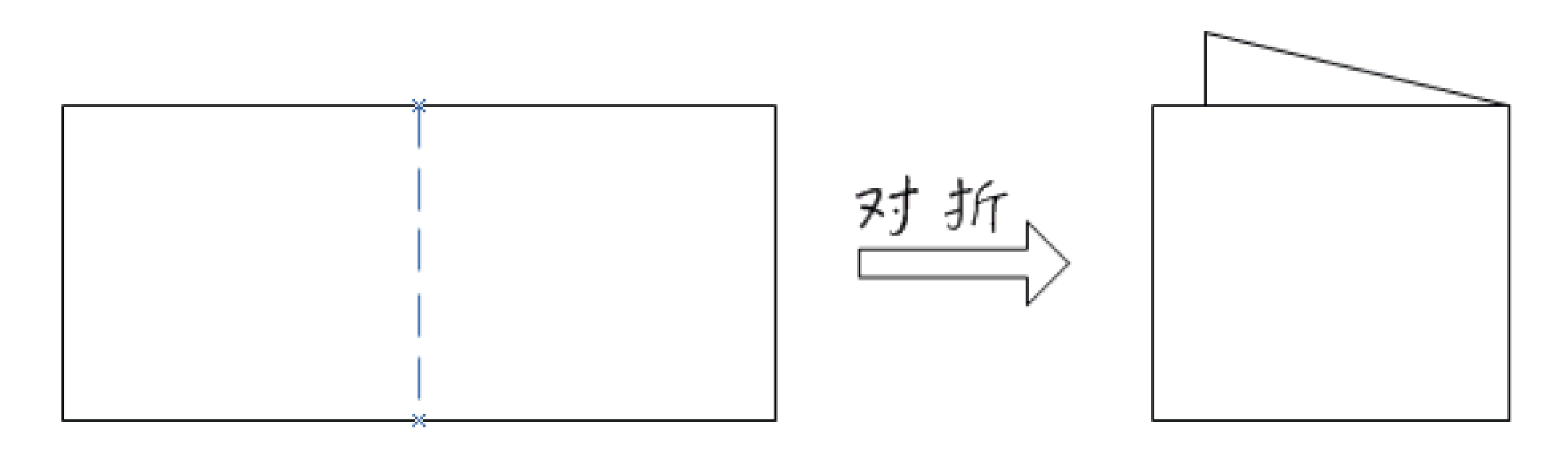


图 7-3

是不是真的能对折任意多次，或对折 10 次以上呢？

我们来试一下。如图 7-4（a）所示，首先将一张纸展平，然后对长边进行对折，得到右图所示效果。这样，对挤后的纸张厚度为 2 张纸的厚度。

接着，如图 7-4（b）所示，将 2 张纸再次沿同一方向进行对折，得到右图所示效果，对折后得到 4 张纸的厚度。

然后，如图 7-4（c）所示，将 4 张纸再次沿同一方向进行对折，得到右图所示效果，对折后得到 8 张纸的厚度。

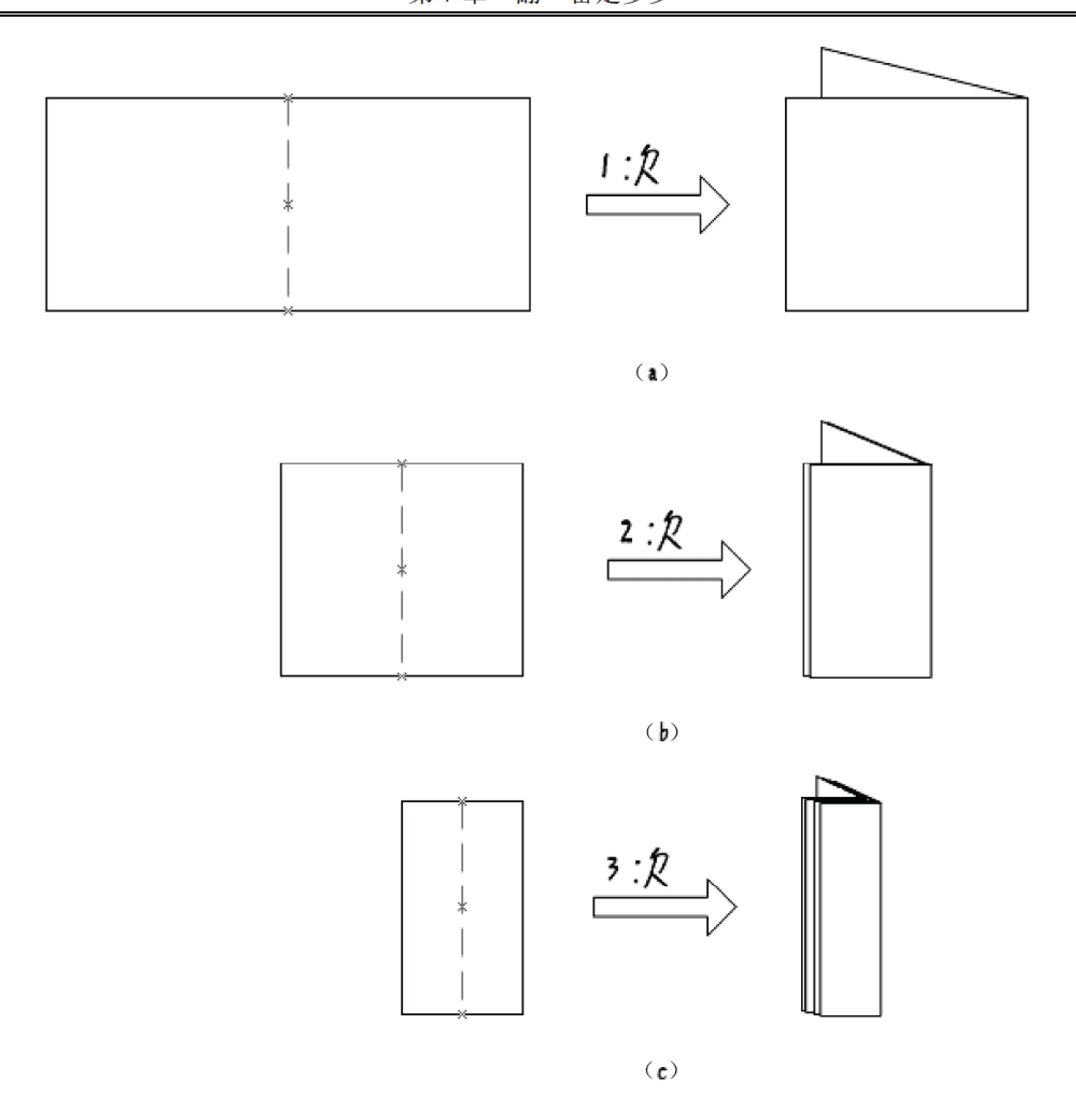


图 7-4

从图 7-4 中可以看到，经过 3 次沿同一方向对折后，对折边的长度已经只有原来长度的八分之一了。

这时可继续沿同一方向进行多次对折，当这一方向不能再对折时，接着沿对折后的长边再次进行对折。一共能对折多少次呢？

经过实验，对于常见的 A4 幅面的纸张，长宽方向都进行对折，对折次数最多达到 7 次。对于幅面很大的报纸（如对开的《人民日报》，其展开长度为 841mm，宽度为 594mm），对折次数最多达到 9 次。

读者可以马上拿一张纸来对折试试！

7.3.2　100 米长的纸能对折几次

通过前面的试验我们可以发现，每对折一次，这叠对折后的纸的张数就会翻番，也就是说这叠纸的厚度会翻番。

并且，当这叠纸的厚度接近对折边的长度时，就没办法进行对折操作了（从理论上来说，对折边的长度应大于厚度，才能进行对折。但实际操作时，当厚度小于并接近对折边的长度时，就没办法进行对折了）。

有了以上分析，除了实际验证之外，我们也可以通过数学的方法，通过计算来进行验证。只需要计算每次对折后的厚度、对折边的长度，就可以方便地判断出是否还能继续对折，也就可以得到对折的次数了。下面，我们就来进行计算。

首先，计算 A4 幅面纸张能对折多少次。

已知 A4 幅面纸张的长为 297mm，宽为 210mm，每平方米 70 克的复印纸的厚度约为 0.08mm。根据这个已知条件，我们来进行计算，每对折一次其对折边长度减半，而厚度翻番。具体计算过程如表 7-8 所示。

表 7-8　A4 纸对折过程

对　折　次　数	长　　度	宽　　度	厚　　度
0	297	210	0.08
1	148.5	210	0.16
2	74.25	210	0.32
3	37.125	210	0.64
4	18.5625	210	1.28
5	9.28125	210	2.56
6	4.640625	210	5.12
7		105	10.24
8		52.5	20.48
9		26.25	40.96

按表 7-8 中的计算，当按 A4 纸的长边（边长为 297mm 这边）连续进行 6 次对折后，其厚度已达到 5.12mm，而长度只有 4.64mm，因此，这时就无法再沿这边进行对折了。从第 7 次对折开始就沿 A4 纸的短边（边长为 210mm）进行对折，第 9 次对折后，纸的厚度已达到 40.96mm，而长度只有 26.25mm 了。显然没办法继续进行对折了。

也就是说，从理论上来讲，A4 纸最多只能对折 9 次。在实际操作中，对折时重叠在一起的纸张厚度之间会有一些间隙，要占用一定的厚度空间，并且对折时需要足够的长度才能使一叠纸弯曲等原因，肯定达不到理论上对折 9 次的效果。通常 A4 纸只能对折 7 次！

从表 7-8 中可看到，纸张的厚度是影响对折次数的一个关键因素，那么，我们找很薄的纸张进行对折，对折次数是不是能明显增加呢？我们再选择 A4 幅面的薄纸来试一下。

每平方米 16 克的薄页纸的厚度为 0.036mm，假设我们使用 A4 幅面的这种纸来进行对折，能对折多少次呢？计算过程如表 7-9 所示。

表 7-9　A4 薄纸对折过程

对　折　次　数	长　　度	宽　　度	厚　　度
0	297	210	0.036
1	148.5	210	0.072
2	74.25	210	0.144
3	37.125	210	0.288
4	18.5625	210	0.576
5	9.28125	210	1.152
6	4.640625	210	2.304
7	2.3203125	210	4.608
8		105	9.216
9		52.5	18.432
10		26.25	36.864

从表 7-9 中可看出，虽然 16g 纸张的厚度比 70g 纸张的厚度少了一半多，但对折次数并没有增加 1 倍，而仅仅增加了 1 次！

按照以上分析，似乎当纸的长度足够长时，对折的次数就可以为任意多次。那么，这个结论正确吗？我们来验证一下。

长度为 297mm 的纸张只能对折 9 次（理论上），我们将长度明显增加，增长到 100m（即 100000mm），相对于 A4 纸张来说，长度方向增加了 300 多倍，对折次数能增加多少呢？还是看实际计算过程吧，如表 7-10 所示。

从表 7-10 中的计算结果可看出，虽然纸张的长度增加了 300 多倍，但是对折次数并没有显著增加，只增加了 1～2 次而已。

在表 7-10 中，计算了两种厚度，一种是按单张纸厚 0.1mm 进行计算，一种是按单张纸厚 0.036mm 进行计算。

表 7-10　长度 100m纸张对折过程

对　折　次　数	长　　度	厚　　度	厚　　度
0	100000	0.1	0.036
1	50000	0.2	0.072
2	25000	0.4	0.144
3	12500	0.8	0.288
4	6250	1.6	0.576
5	3125	3.2	1.152
6	1562.5	6.4	2.304
7	781.25	12.8	4.608
8	390.625	25.6	9.216
9	195.3125	51.2	18.432
10	97.65625	102.4	36.864
11	48.828125		73.728

从上面的分析可以看到，对折的关键还是在于纸张的厚度，每对折一次，纸的厚度值就翻一番。根据本章前面介绍的翻番知识，可用以下公式计算对折后的厚度：

$$对折n次厚度 = 单张纸厚度 \times 2^n$$

根据以上公式，可以计算出对折 *n* 次后纸的厚度是单张纸厚度的多少倍，如表 7-11 所示。

表 7-11　对折几次后厚度对比表

次数	厚度	次数	厚度	次数	厚度	次数	厚度
1	2	11	2048	21	2097152	31	2147483648
2	4	12	4096	22	4194304	32	4294967296
3	8	13	8192	23	8388608	33	8589934592
4	16	14	16384	24	16777216	34	17179869184
5	32	15	32768	25	33554432	35	34359738368
6	64	16	65536	26	67108864	36	68719476736
7	128	17	131072	27	134217728	37	1.37439E+11
8	256	18	262144	28	268435456	38	2.74878E+11
9	512	19	524288	29	536870912	39	5.49756E+11
10	1024	20	1048576	30	1073741824	40	1.09951E+12

从表 7-11 中可看出，当对折 40 次后，得到的厚度为单张纸厚度的 1.09951×10^{12} 倍。假设单张纸的厚度为 0.1mm，则对折 40 次后的厚度为：

$$0.1 \times 1.09951 \times 10^{12} = 1.09951 \times 10^{11} (mm)$$

$$= 109951 \ (km)$$

也就是说，将厚度为 0.1mm 的纸张对折 40 次，得到对折后纸张的厚度将达到 10 万公里以上，可以绕地球 17 圈！

7.3.3　计算对折次数的程序

对于纸张的对折，如果每改变一次长度、宽度或纸张的厚度，我们都需要重新去推算一次，会比较麻烦。作为程序员，首先想到的就是编写一个程序，能快速计算出理论上的对折次数。这个程序比较简单，只需要重复计算对折边长、对折厚度，然后进行判断即可。具体程序如下：

```
#include <stdio.h>

int main()
{
    float thickness;                        //单张纸的厚度
    float total;                            //总的厚度
    float side[2];                          //纸的长度和宽度
    int count;                              //对折次数

    int i;
    float s;                                //对折边长度
```

```
    printf("纸张的长度: ");
    scanf("%f",&side[0]);
    printf("纸张的宽度: ");
    scanf("%f",&side[1]);
    printf("纸张的厚度:");
    scanf("%f",&thickness);

    total=thickness;
    count=0;
    printf("\n 次数\t 对折边长\t\t 厚度\n") ;
    for(i=0;i<2;i++)
    {
        s=side[i];                              //取得边长
        if(i==0)
            printf("从长度方向对折\n");
        else
            printf("从宽度方向对折\n");
        while(s>total)
        {
            count++;
            s/=2;                               //对折边长减半
            total*=2;                           //厚度加倍
            printf("%2d\t%10.2f\t\t%10.2f\n",count,s,total);
        }

    }

    getch();
    return 0;
}
```

在以上程序中，由于在长度或宽度方向进行对折的操作都是相同的，因此使用一个数组来保存长度和宽度，方便程序中使用循环处理对折。

编译并运行以上程序，参照表 7-8 所示的数据，输入纸张的宽度为 297、宽度为 210、厚度为 0.08，则可得到如图 7-5 所示的运行结果。

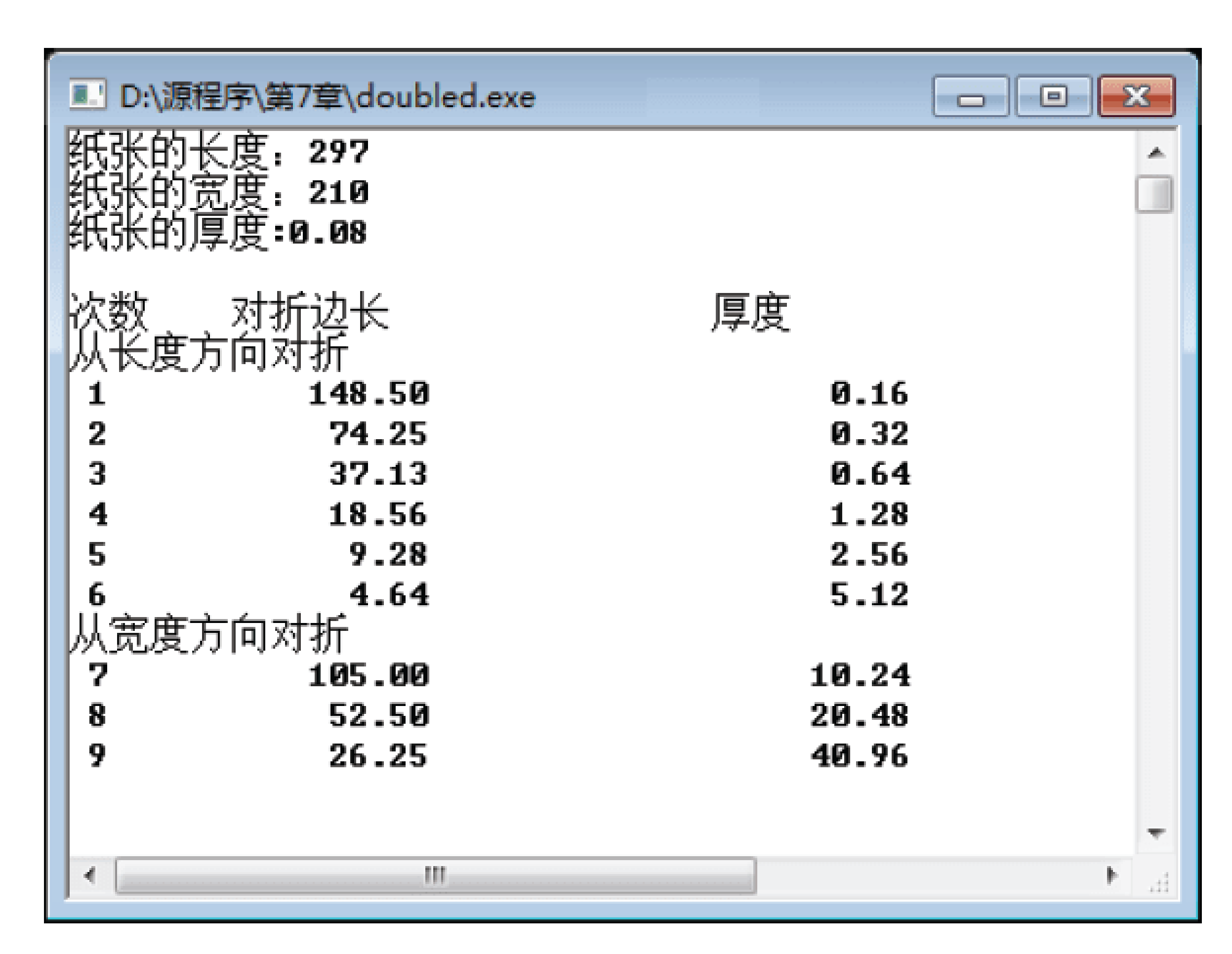


图 7-5

7.4　一棋盘的麦子

直觉告诉我们，长度达 100m 的纸张对折的次数应该比长度为 0.297m 的纸张对折的次数多得多。但是，我们的直觉出错了，通过 7.3 节的介绍可以看出，长度为 0.297m 的纸张沿长度方向可以对折 7 次（见表 7-8），而长度为 100m 的纸张沿长度方向可以对折 10 次（见表 7-9）。显然，长度增加了 300 多倍，但对折次数只增加了 3 次。

为什么会出现这种错觉呢？这是由于我们大部分时间里都是以线性方式来考虑问题，而这类翻番的问题，不是线性的问题。

下面再来看一个古老的问题，一个印度饶有趣味的故事。

7.4.1　舍罕王的赏赐

据说古代印度的舍罕王打算给国际象棋的发明人、印度的宰相西萨・班・达依尔以重重的赏赐。国王问他有什么要求，将尽量满意他的要求。

这位宰相意味深长地抿嘴笑着，跪在国王面前，指着放在国王膝盖上的国际象棋棋盘说道："陛下，请您在这张棋盘的第 1 个小方格里给我 1 粒麦子，在第 2 个小方格里给 2 粒，第 3 格里给 4 粒，照这样下去，每 1 小格里都比前 1 小格的数量加 1 倍（如图 7-6 所示）。陛下啊，这样将棋盘上所有 64 个小方格的麦粒，都赏赐给您的仆人吧！"





图 7-6

国王一听，认为这区区赏赐，微不足道。于是，满口答应道："爱卿，你所求的并不多啊，你当然会如愿以偿的。"

国王为自己对这样一件奇妙的发明所许下的慷慨赏诺不致破费太多而暗暗高兴。他令人把一袋麦子拿到座前，开始往国际象棋棋盘上放置麦粒了。在第 1 格里放 1 粒，第 2 格里放 2 粒，第 3 格里放 4 粒，这样放到第 20 格，1 袋麦子就空了，到第 21 格时，1 袋麦子已经不够了！

接下去麦粒的数量 1 格接 1 格飞快地增长下去，1 袋又 1 袋的麦子被扛到国王面前来，堆成了小山，可是棋盘上的格子还多着哩。

舍罕王愣了，他发现：即使拿来全印度的粮食，也不能兑现他对西萨•班•达依尔宰相许下的诺言了。

7.4.2　需要多少麦粒

那么，究竟需要多少麦粒才能完成舍罕王的赏赐呢？下面我们来计算一下。根据西萨•班•达依尔的要求，在棋盘的 64 个格子中，第 1 个格中放置 1 粒小麦，然后每个格子中的麦粒数量比前一个格子的麦粒数翻一番。

下面先计算前 20 个格子中每个格子的麦粒数量，具体如表 7-12 所示。可以看出，在棋盘前 10 个格子中，每个格子中的麦粒数都在 1000 粒以内。但是，随着麦粒数量的不断翻番，到第 20 个格子，麦粒数已超过 50 万粒了！

表 7-12　每个格子的麦粒数量

第 1 格	1	第 11 格	512×2=1024
第 2 格	1×2=2	第 12 格	1024×2=2048
第 3 格	2×2=4	第 13 格	2048×2=4096
第 4 格	4×2=8	第 14 格	4096×2=8192
第 5 格	8×2=16	第 15 格	8192×2=16384
第 6 格	16×2=32	第 16 格	16384×2=32768
第 7 格	32×2=64	第 17 格	32768×2=65536
第 8 格	64×2=128	第 18 格	65536×2=131072
第 9 格	128×2=256	第 19 格	131072×2=262144
第 10 格	256×2=512	第 20 格	262144×2=524288

其实，我们也可以不用这种表格来推算，而是用一个公式直接计算出棋盘的某个编号的格子应该放置多少粒麦粒，具体计算公式如下：

$$\text{第 } n \text{ 格麦粒数} = 2^{n-1}$$

有了以上公式，就可直接计算第 64 格中应该放置多少麦粒了：

$$\text{第 64 格麦粒数} = 2^{64-1} = 9223372036854775808$$

这个数太大了！具体有多大呢，我们将这些麦粒数量换算成重量来衡量一下。查询了一些资料，通常小麦千粒重 35～40 克，也就是说 1 公斤小麦大概有 25000～28000 粒。如果按 28000 粒小麦为 1 公斤，则第 64 格中放置的麦粒重量为：

$$9223372036854775808 \div 28000 \div 1000 = 329406144173(\text{吨})$$

仅棋盘这一格就需要 3000 多亿吨小麦，如果将棋盘中所有 64 格中的小麦数量累加起来，将需要更多的小麦。达依尔所要求的竟是全世界在两千年内所生产的全部小麦！

这么一来，舍罕王发现自己欠了宰相很大一笔债。要么是忍受西萨•班•达依尔没完没了的讨债，要么是干脆砍掉他的脑袋。国王大概选择了后面的这个办法。

舍罕王为什么会吃这样的亏呢？因为他根本没有这么巨大数量的感性认识，即使比他经验丰富、知识广博的现代人，也不能一下子直接觉察到这个数量有多大。

这就是翻番的威力！只有认识了翻番的威力，才能把握到数据快速增加的诀窍！

那么，如果按照西萨·班·达依尔的要求，最后究竟需要多少麦粒，折合成重量是多少吨呢？下面编写程序，借助计算机来帮助我们计算出最终结果。

```
#include <stdio.h>

int main()
{
    double chessboard,sum;
    int i;

    chessboard=1;
    sum=1;
    for(i=2;i<=64;i++)
    {
        chessboard*=2;
        sum+=chessboard;
        printf("第%d 格，麦粒：%10.0lf\n",i,chessboard);
    }
    printf("\n 总麦粒数：%10.0lf,约合：%10.2lf 吨\n",sum,sum/28000/1000);

    getch();
    return 0;
}
```

这个程序比较简单，由于计算的麦粒数是一个非常大的数据，因此定义一个 double 变量来保存，这样将只能得到一个大约数据，麦粒数并不能精确到个位。

编译运行以上程序，得到如图 7-7 所示的结果。从运行结果可以看出，总的麦粒数约为第 64 格的麦粒数的 2 倍，折算成重量也约等于第 64 格的麦粒的 2 倍。

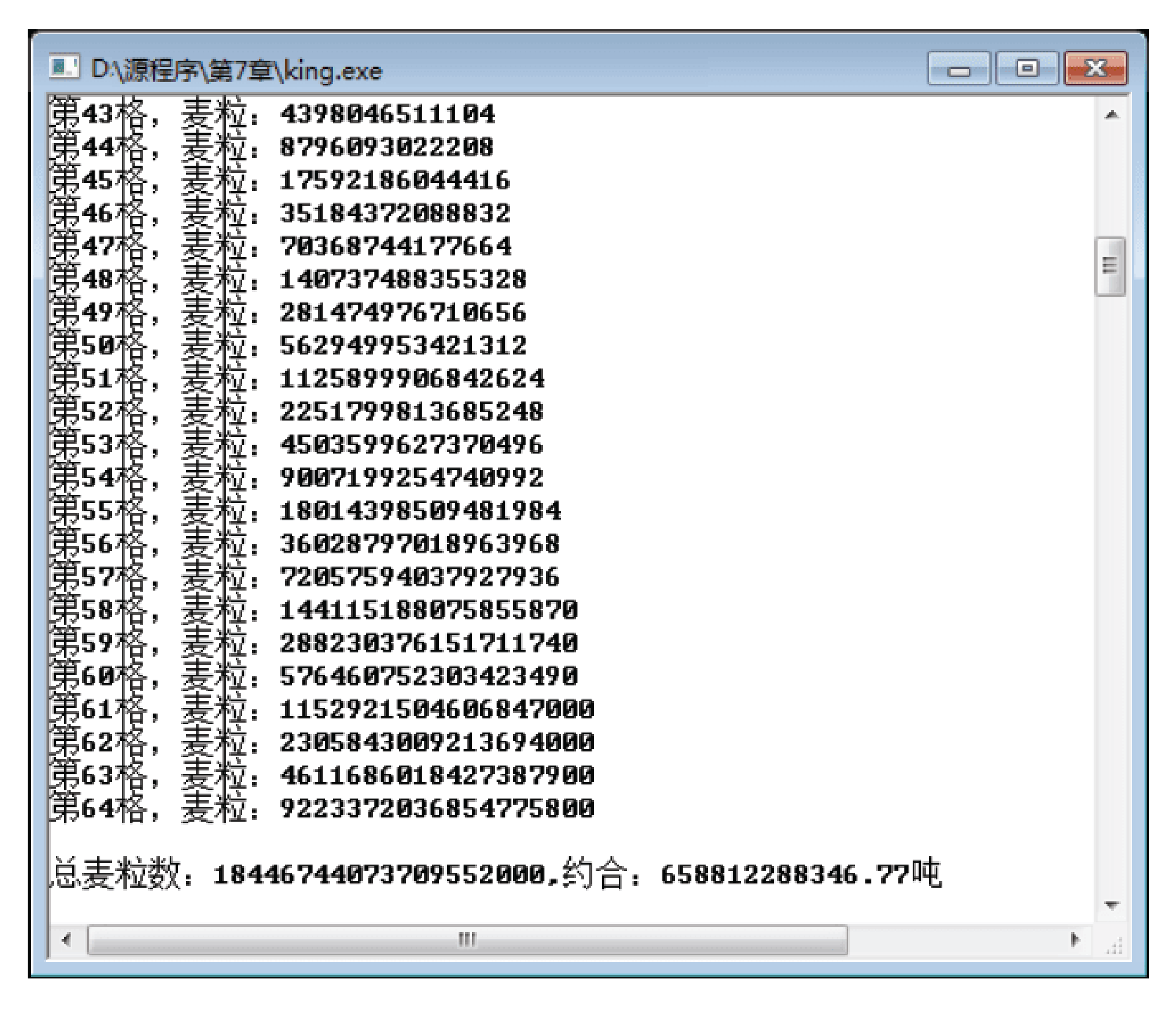


图 7-7

7.5　折半法的运用

通过前面的例子，我们已经认识了翻番的威力：一个很小的数，经过多次翻番后，很快就会得到一个非常大的数，如 7.4 节中舍罕王的赏赐，经过 63 次翻番，从最初的数字 1 增长成一个天文数字。通过这些例子，我们已经知道翻番的运算是很简单的，就是将前期的数据乘以 2，即可得到本期数据。

7.5.1　翻番的逆运算

翻番的逆运算也很简单，就是将前期的数据除以 2，用来得到本期数据，这种计算方法称为折半法，也称为二分法。

翻番运算：前期数据 × 2 = 本期数据

折半运算：前期数据 ÷ 2 = 本期数据

根据翻番运算的规则我们可以推出，对于原本很大的数据，通过多次折半运算后，可以得到一个较小的数据。

在实际应用中，将某一个很大的数进行折半运算以得到一个较小数据的情况不是太多。不过，在程序设计中，经常会用到一种查找算法：折半查找法。这种算法就是将原来需要处理的数量很庞大的一组数据，通过折半运算不断缩减运算规模，以方便、快速地查找出需要的数据。

例如，原来要在 10 亿组数据中查找某一个数，如果逐个去比较这些数据，最坏的情况下，需要进行 10 亿次比较才能找到需要的数据。而如果使用折半运算，第一次折半运算就可以将问题规模缩减为 5 亿，再次折半运算又可缩减为 2.5 亿，经过多次折半运算，很快就可以问题规模缩减到个位数，并找到目标数据。

因此，在程序设计中，折半运算在数据查找类运算中应用十分广泛。

7.5.2　找出假硬币

下面我们来看一个小学生常见的智力题：找假硬币。

现有 100 枚硬币，知道其中有 1 枚是假的，不过，由于假硬币的外观与真硬币完全一样，凭肉眼无法分辨。但是由于使用的材质不同，假硬币的重量比真硬币的重量轻。现在用一个天平作为工具，从这 100 枚硬币中找出那 1 枚假硬币，问：最少经过多少次称重必定能找出那枚假硬币？

看起来这个问题很简单，就是找出假硬币，但是，要求用最小的次数且必须找出假

硬币。

在这道题中，只有天平这一个工具可以使用，因此，只能通过称重的方法来找出假硬币。虽说只有称重这一方法，可是怎么称重又可以分很多种情况。

1. 逐个称重

要通过天平这个工具找出假硬币，最简单易懂的方法就是：用天平逐个称出每枚硬币的重量并记录下来，然后从这些记录中找出重量最轻的那枚就是假硬币。

序号	重量	
1	6	
2	6	
3	6	
4	6	
5	6	
6	6	
7	6	
8	5.8	轻，假硬币
……	……	

最理想的情况是，称第 1 枚和第 2 枚重量时，就发现这两枚硬币的重量不一样，这时就可知道重量轻的那枚为假硬币。也就是说，通过称重 2 次就可找出假硬币。

最坏的情况是，对第 100 枚硬币进行称重时才发现其重量比其他硬币轻，即需要经过 100 次称重才能找出假硬币。

显然，这种方法的效率不会让人满意。

2. 折半称重

根据前面提到的折半运算方法可将问题规模快速缩减，对于这个问题，就可考虑使用折半运算的方法，即将 100 枚硬币进行折半运算，可分为两部分，每一部分为 50 枚，然后分别对这两部分硬币进行称重。可以知道，重量轻的那部分中就包含有假硬币。接着对包含假硬币这部分（共 50 枚硬币）再次进行折半运算，又分为两部分，每一部分 25 枚，再通过称重找出假硬币在哪一部分并不断重复进行折半运算，最终即可找出假硬币。

由于使用天平进行称重，因此，可以将折半运算分成的两部分硬币分别放在天平的两边。也就是说，每次折半运算，只需要称重一次就可发现假币在哪一部分中，不需要对折半分出的两部分硬币分别称重。这样，可减少称重的次数。

根据以上思路，折半称重的过程如图 7-8 所示

从图 7-8 中可看出，经过 6 次称重，必定能找出那枚假硬币。在最理想的情况下，第 3 次称重就可找出那枚假硬币。

第 1 次称重时，将 100 枚硬币分成两部分，天平的左边放 50 枚，右边放 50 枚。如

果天平右边秤盘中的硬币较轻，说明那枚假硬币位于右边秤盘中。

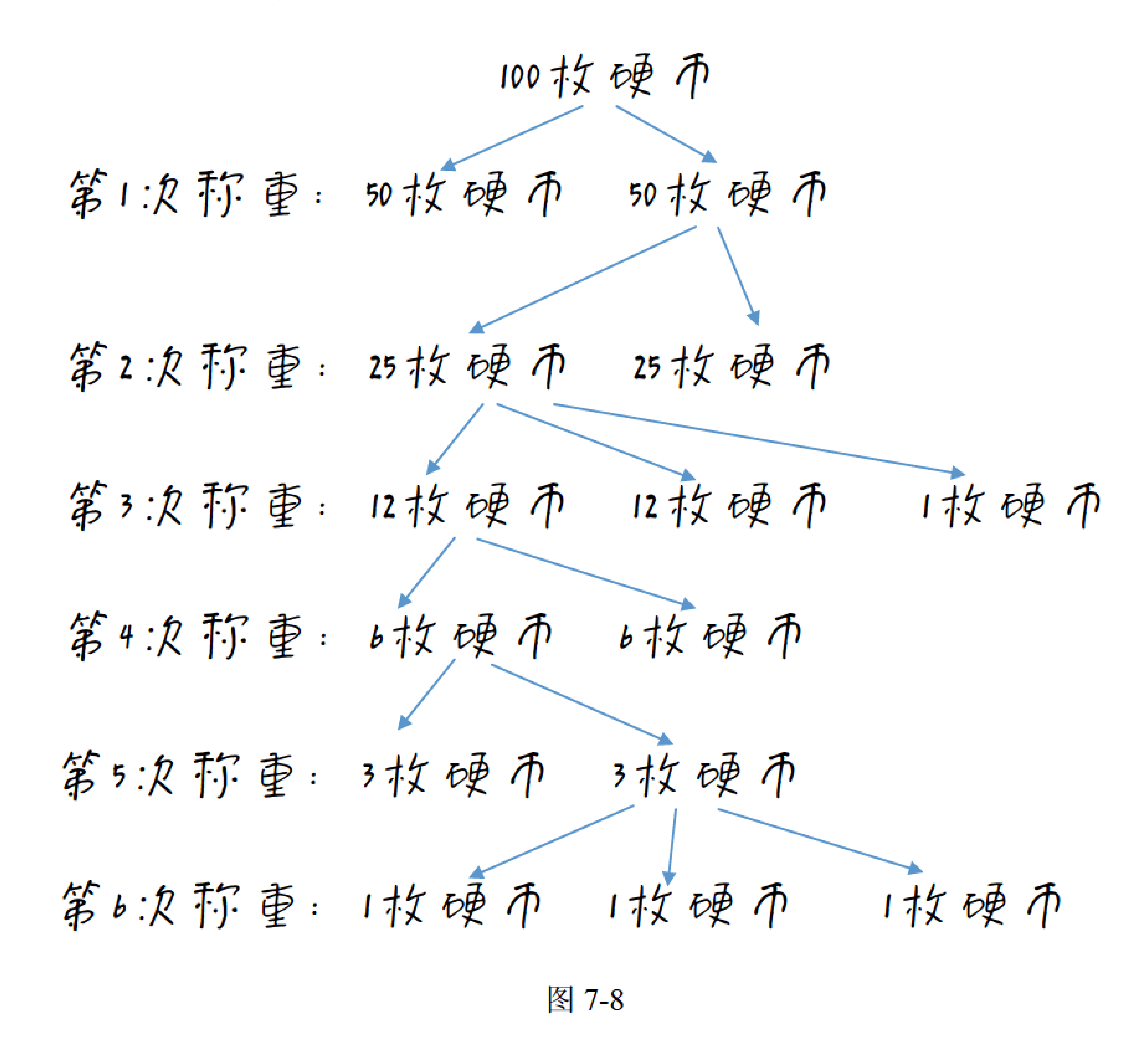


图 7-8

第 2 次称重时，将右边秤盘中的 50 枚硬币又分成两部分，左边放 25 枚，右边放 25 枚。如果天平左边秤盘中的硬币较轻，说明那枚假硬币位于左边秤盘中。

第 3 次称重时，由于 25 枚硬币不能进行平均折半分到两个秤盘中，这时可在左边放 12 枚，右边放 12 枚，剩余 1 枚。这次称重就会有 3 种情况：第一，如果天平秤盘两边的重量相等，说明剩余的 1 枚为假硬币，完成称重；第二，如果右盘秤盘中的硬币较轻，说明那枚假硬币位于右边秤盘中；第三，如果左边秤盘中的硬币较轻，说明那枚假硬币位于左边秤盘中。

第 4 次称重，若第 3 次称重未找到假硬币，则将假硬币所在秤盘中的 12 枚硬币再次折半分到两个秤盘中，左边放 6 枚，右边放 6 枚，如果天平左边秤盘中的硬币较轻，说明那枚假硬币位于左边秤盘中。

第 5 次称重，若第 4 次称重未找到假硬币，将 6 枚硬币折半分到两个秤盘中，左边放 3 枚，右边放 3 枚，如果天平右边秤盘中的硬币较轻，说明那枚假硬币位于左边秤盘中。

第 6 次称重，若第 5 次称重未找到假硬币，与第 3 次类似，将 3 枚硬币分成 3 份，左边放 1 枚，右边放 1 枚，剩余 1 枚。这次称重就会有 3 种情况：第一，如果天平秤盘两边的重量相等，说明剩余的 1 枚为假硬币；第二，如果右盘秤盘中的硬币较轻，说明右边秤盘中的为假硬币；第三，如果左边秤盘中的硬币较轻，说明左边秤盘中的为假

硬币。

3. 三分称重

在图 7-8 所示折半称重的方法中，第 3 次和第 6 次称重时，都是将硬币分为了 3 部分，如果在天平左右秤盘上的两部分相等，则说明假硬币在第 3 部分中。那么，如果我们每次都将硬币分为 3 部分进行处理，会不会减少称重的次数呢？

答案是：在特定的条件下，这种三分称重可以减少称重次数。

具体的特定条件是什么呢？就是硬币的数量为 3^n 时，可使用这种三分称重法。

例如，当 80 枚真硬币中混入 1 枚假硬币（共 81 枚硬币）时，就可使用这种三分称重法，具体过程如图 7-9 所示。

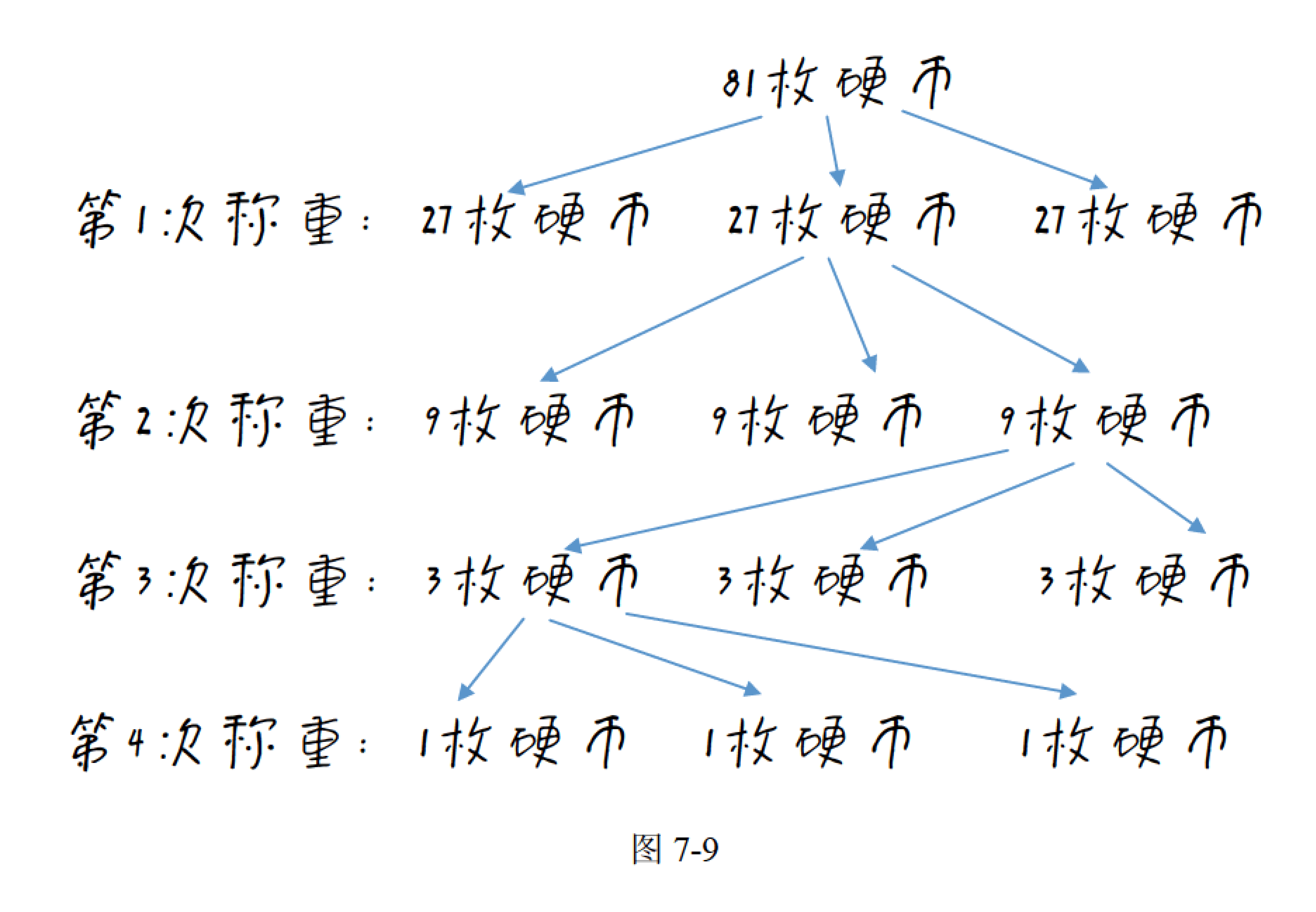


图 7-9

从图 7-9 中可看出，对于 81 枚硬币，当采用三分称重法时，只需要称重 4 次就可找出假硬币

第 1 次称重时，将 81 枚硬币均分为 3 部分，天平左边放 27 枚，右边放 27 枚，剩余 27 枚。这时将有 3 种情况：第一，天平两边的硬币重量相等，则假硬币在剩余的 27 枚中。第二，天平左边硬币重量较轻，则假硬币位于左边 27 枚中。第三，天平右边硬币重量较轻，则假硬币位于右边 27 枚中。

第 2 次称重时，将 27 枚硬币均分为 3 部分，天平左边放 9 枚，右边放 9 枚，剩余 9 枚。同样按 3 种情况来确定假硬币所在的部分。

第 3、4 次称重时类似，都是将假硬币所在部分均分为 3 部分，其中 2 部分放在天平左右秤盘中，1 部分剩余。直到折半到只有 1 枚硬币时就可直接找出假硬币了。

如果按折半方法来查找假硬币，则其查找过程如图 7-10 所示，与 100 枚硬币相似，最多需要 6 次称重。在理想情况下，称重 1 次，就可找出假硬币（为剩余的那枚）。

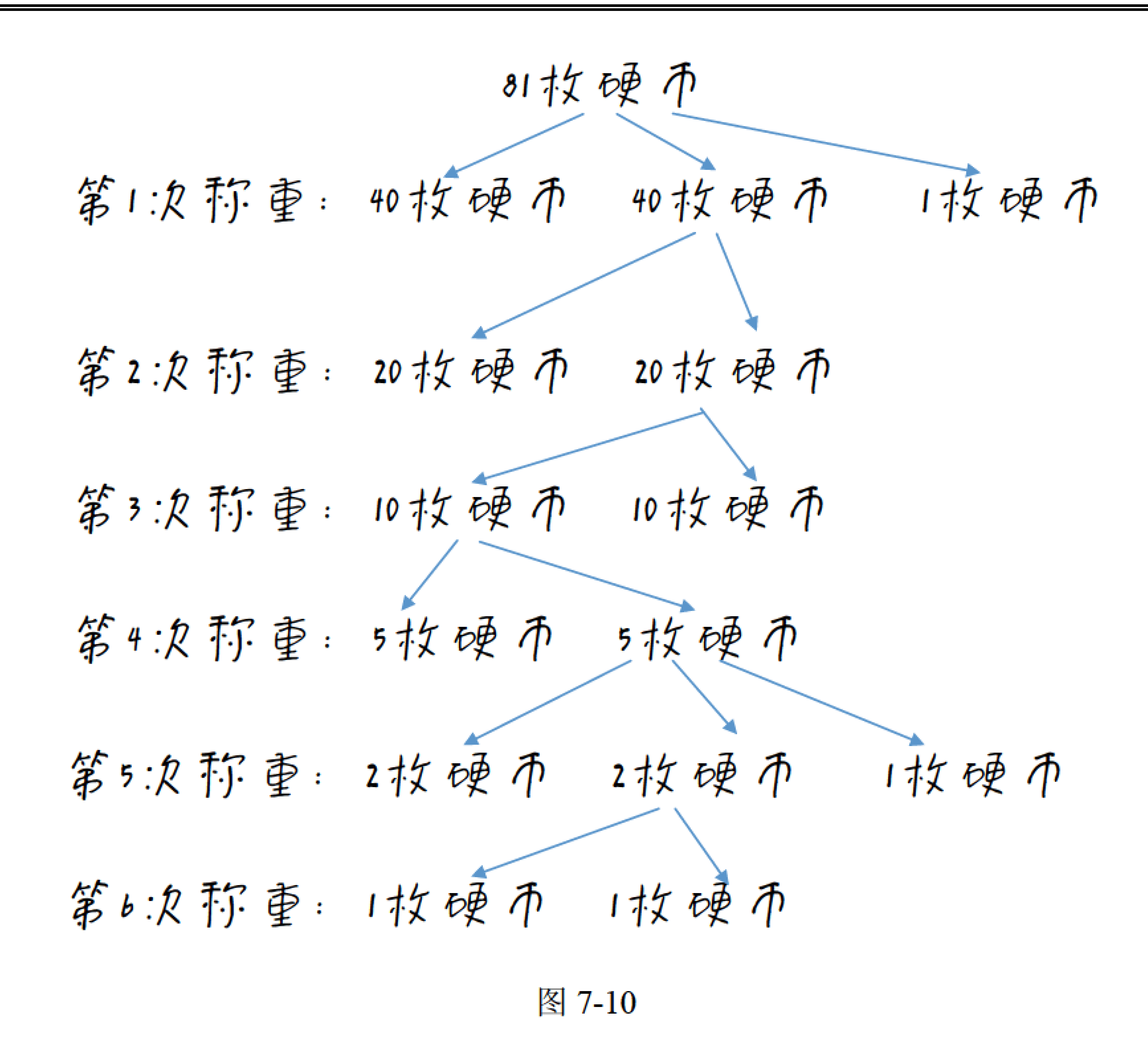


图 7-10

虽然使用三分称重的方法可以减少称重的次数，但那是有条件的，即硬币的数量需要为 3^n。对于不满足这个条件的情况，则没办法使用这种方法。

7.5.3　编写程序找出假硬币

前面我们介绍了用称重法找出假硬币的过程，可以看出，使用折半运算是最方便、快速的方法（虽然用三分称重法可减少称重次数，但有特殊的限制条件）。

下面，我们编写一个程序来模拟这种折半称重。在程序中，通过随机函数模拟生成一枚假硬币，并使其重量比真硬币轻。然后程序模拟折半称重的过程，对任意数量的硬币进行模拟操作，并找出假硬币所在序号，具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#define NUMBER 100

int main()
{
    int coin[NUMBER];                //保存硬币重量
    int i,j,s;
    int low,high,mid,total1,total2; //保存每次称重后的数组下标序号、重量
    int count=0;                     //计数
    int fake=-1;                     //假硬币的下标

    srand((int)time(NULL));          //生成随机数种子
```

```
    for(i=0;i<NUMBER;i++)              //将硬币全部设置为相同重量
    {
        coin[i]=6;
    }

    s=rand()%NUMBER;//生成一个随机数表示假硬币的下标
    coin[s]=5;                         //对应位置为假硬币重量（小于真硬币重量）
    //printf("假硬币的下标:%d\n",s);   //输出假硬币下标

    low=0;                             //假硬币范围下限
    high=NUMBER;//假硬币范围上限
    mid=(high-low)/2+low;              //折半位置

    while(low<=mid-1)
    {
        count++; //称重次数
        if((high-low)%2==1)            //折半后不为整数
        {
           fake=--high;                //设最后一枚硬币为假币
        }

        total1=total2=0;               //重量初始化
        for(j=low;j<mid;j++)           //累加左秤盘硬币重量
        {
            total1+=coin[j];
        }
        for(j=mid;j<high;j++)          //累加右秤盘硬币重量
        {
            total2+=coin[j];
        }

        if(total1<total2)              //左秤盘硬币重量轻
        {
            high=mid;                  //缩小折半范围
            fake=high-1;               //保存可能的假硬币下标
        }else if(total1>total2)        //右边秤盘硬币重量轻
        {
            fake=low=mid;              //缩小折半范围，保存可能的假硬币下标
        }else                          //两边秤盘重量相等
        {
            break;                     //退出循环，完成称重
        }

        printf("第 %d 次称重后，假币位于 %d～%d 之间\n",count,low,high);
        mid=(high-low)/2+low;          //进行折半运算
    }

    printf("共进行了 %d 次称重，第 %d 枚硬币是假的！\n",count,fake+1);

    getch();
    return 0;
}
```

以上代码右侧有详细的注释，可看出程序是完整模拟折半称重的过程。编译运行以

上程序，可看到如图 7-11 所示的运行结果。当然，由于程序中使用随机序号作为假硬币的序号，因此，每次运行的结果都不一样。要查看程序运行结果是否正确，应将上面程序代码中注释掉的那一行启用，以显示生成的随机序号数值，用来和最终称重结果进行比对，看程序运行是否正确。在图 7-11 中，随机函数生成的序号是 60，经过程序运算找出的也是第 60 枚为假硬币——由于 C 语言数组是从 0 开始的，因此，这里其实是第 61 枚硬币为假的，所以输出时将其加 1，显示为 61（表示在 1～100 枚硬币之间，第 61 枚硬币为假硬币）。



图 7-11

7.5.4 折半法在查找中的应用

其实，折半法在查找算法中应用得更多，在查找算法中折半查找又称为二分查找。对于大批量数据，使用折半查找可以大幅提高查找的效率。

需要注意的是，要使用折半查找算法，被查找的数据必须要满足以下两个条件：

- ❑ 被查找的数据是线性结构保存的。
- ❑ 被查找的数据是按关键字有序排列的。

折半查找算法的具体过程如下：

假设有 n 个元素的查找表（要查找的源数据），首先计算位于查找表中间位置元素的序号 m（$m=n/2$），取 s[m]的关键字与给定值 key 进行比较，比较结果有 3 种可能：

（1）若 s[m]=key，表示查找成功，找到所查关键字的位置 m。

（2）若 s[m]>key，表示关键字 key 只可能在查找表的前半部分（因查找表中的数据是按从小到大的顺序排列），则在前半部分继续进行折半查找。

（3）若 s[m]<key，表示关键字 key 只可能在查找表的后半部分，则在后半部分继续进行折半查找。

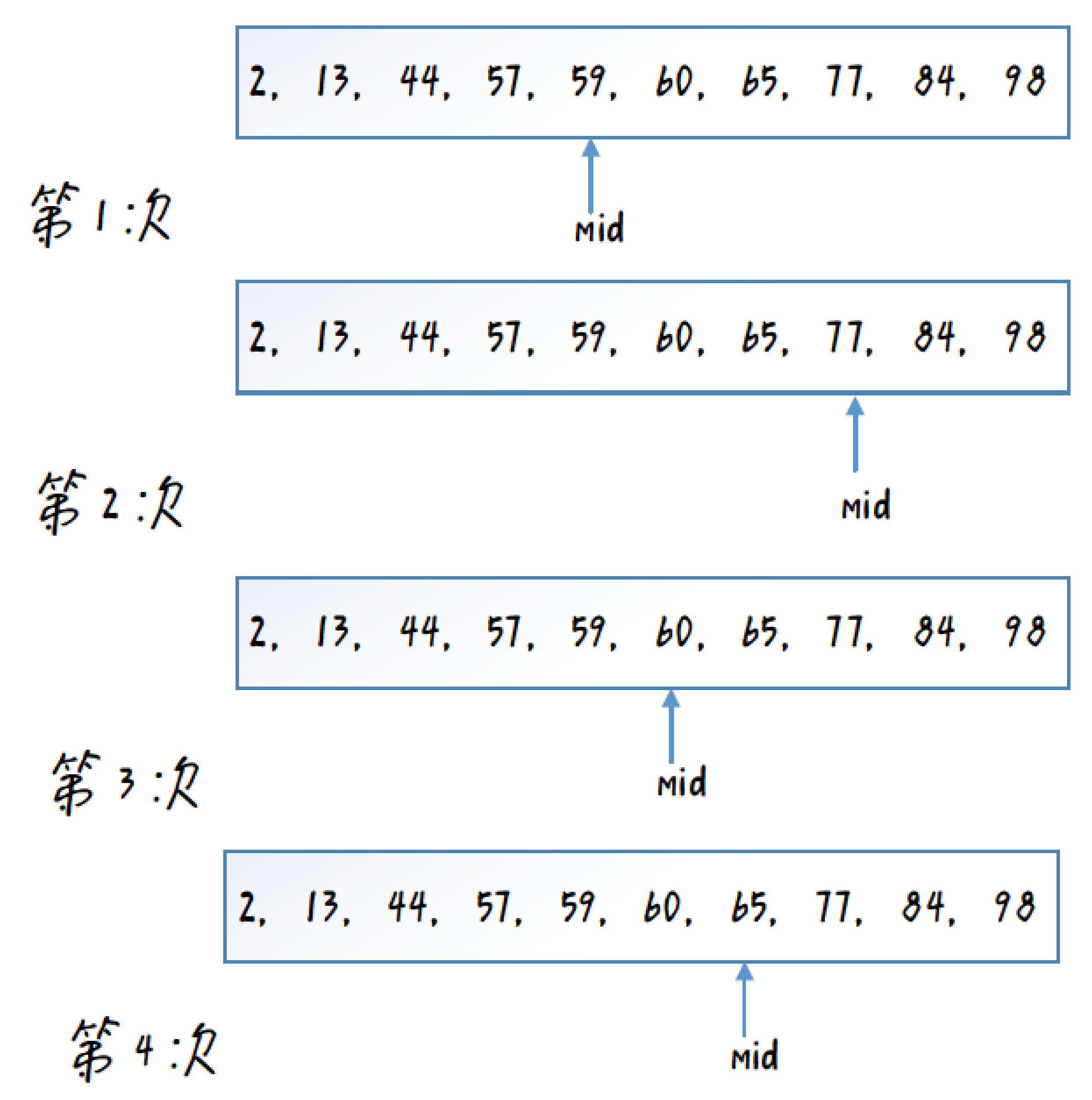
可以看到，与前面进行折半称重类似，每经过一次查找，就将被查找数据的范围缩小一半。通过多次查找后，可将查找范围缩小到一个数据，最终可比较这个数据是否为要查找的关键字。

在查找中，并不能保证所查找的关键字必定在被查找数据中存在，当查找范围缩小到只剩下一个元素，而该元素仍与关键字不相等时，则说明查找失败。

根据前面的描述，我们先手工演示一下查找过程。例如，在以下 10 个数据，要求查找出关键字 65 所在位置的序号。

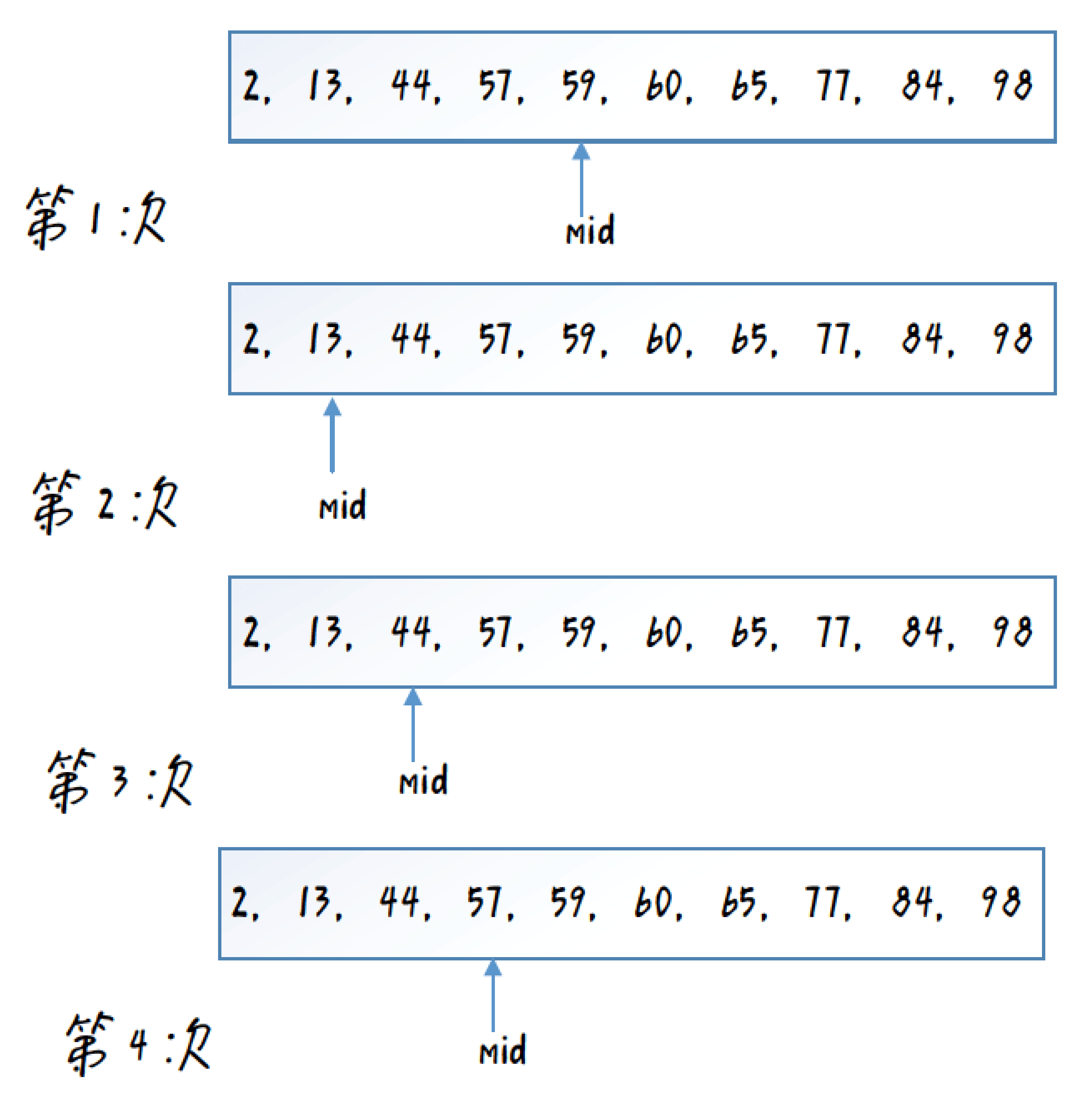
2，13，44，57，59，60，65，77，84，98

用折半查找法查找关键字 65，查找过程如下：



从上面的查找过程可看出，通过 4 次比较，可找到关键字 65 所在的位置。

在上面这 10 个数据中，如果要查找关键字 50 的位置，其查找过程如下：



从上面的过程可看出，通过 4 次比较，当查找范围缩小到只剩一个元素，仍没有找到指定关键字，则说明查找失败。

根据以上对折半查找法的描述，可编写如下折半查找程序。

```
#include <stdio.h>
#define ARRAYLEN 10
```

```
int source[]={2,13,44,57,59,60,65,77,84,98};

int BinarySearch(int s[],int n,int key)
{
int low,high,mid;

   low=0;
   high=n-1;
   while(low<=high)                           //查找范围包含至少一个元素
   {
      mid=(low+high)/2;                       //计算中间位置序号
      if(s[mid]==key)                         //中间元素与关键字相等
         return mid;                          //返回序号
      else if(s[mid]>key)                     //中间元素大于关键字
         high=mid-1;                          //重定义查找范围
      else //中间元素小于关键字
         low=mid+1;                           //重定义查找范围
   }
   return -1;                                 //返回查找失败
}

int main()
{
   int key,i,pos;

   printf("请输入要查找的关键字:");
   scanf("%d",&key);                          //输入查找关键字

   pos=BinarySearch(source,ARRAYLEN,key);     //调用折半查找函数进行查找

   printf("原数据表:");
   for(i=0;i<ARRAYLEN;i++)                    //显示查找表中的数据
      printf("%d ",source[i]);
   printf("\n");

   if(pos>=0)                                 //根据查找结果输出不同的值
      printf("查找成功，该关键字位于数组的第%d 个位置。\n",pos);
   else
      printf("查找失败!\n");

   getch();
   return 0;
}
```

这段程序的代码很简单，与前面折半称重的程序类似，编译运行以上代码，输入查找关键字“65”，将得到如图 7-12 所示的结果。

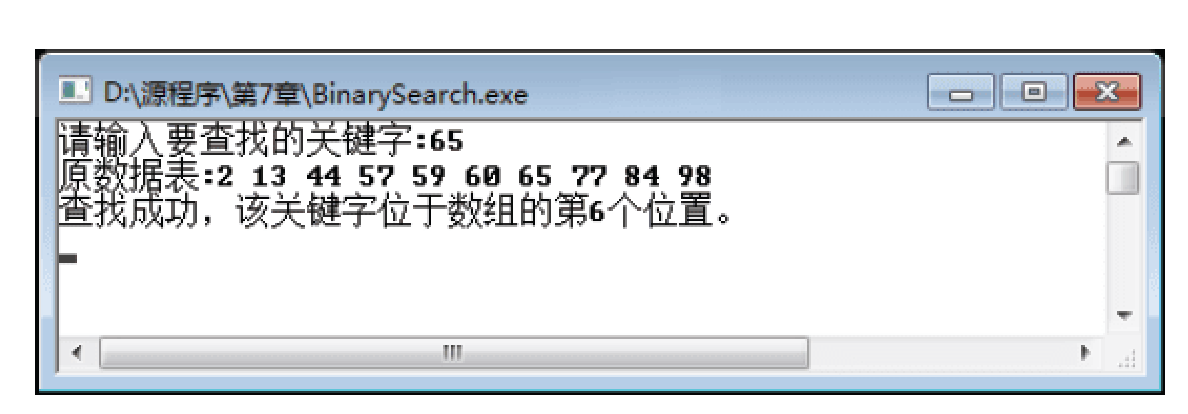


图 7-12

第 8 章　数理逻辑——非此即彼

数理逻辑又称符号逻辑、理论逻辑，它既是数学的一个分支，也是逻辑学的一个分支。数理逻辑是用数学方法研究逻辑或形式逻辑的学科。其研究对象是对证明和计算这两个直观概念进行符号化以后的形式系统。数理逻辑是数学基础的一个不可缺少的组成部分。虽然名称中有逻辑两字，但并不属于单纯逻辑学范畴。

8.1　逻辑的重要性

数理逻辑就是精确化、数学化的形式逻辑，它是现代计算机技术的基础。作为程序员，必须掌握数理逻辑的相关知识。这样，不仅在编写程序时不会出现逻辑错误，并且在编写项目文档等用自然语言描述的信息时，也同样能做到表达清楚，不会产生歧义。

其实，不仅在计算机项目中逻辑非常重要，在日常生活中，逻辑也非常重要。例如，我们编写报告、发言稿、总结等公文，与朋友聊天等活动，都需要有一定的逻辑思维能力，按一定的逻辑关系来进行。

8.1.1　模棱两可的表述

什么叫“模棱两可”？我们先通过一个故事来看这个词的来历。

唐朝时代，栾城有一个人，名字叫苏味道。他九岁的时候就会写文章，后来和他的同乡李峤都以才学出名，当时的人合称他们为苏李。苏味道在二十岁的时候，考取了进士，曾做到吏部侍郎的职位。后来在武则天当皇帝的时候做了宰相。根据“唐书”的记载，苏味道做了宰相以后，只求保持个人的地位和安全，处理事情总是这样办也行，那样办也可以，却从不表示明确的态度和意见，更谈不上什么创新和改革了。他还常常对别人说：“处理事情不能做明确的决断。因为如果发生了错误，就要负失责的责任，只要保持‘模棱’两端就可以了。”当时的人听他这么一说，都叫他“苏模棱”或者是“模棱手”。“模棱”是指方向可左可右的意思。因此，后人在遇到有人说话或处理事情不做明确的决断，也不表示鲜明的态度，可以或不可以都行，就叫做“模棱两可”。

我们来看一下现实生活中一些常见的模棱两可的表述。

案例一

在购买手机时，征求朋友的意见：你看这款手机屏幕很大，玩游戏很好；另外这一款屏幕比较小，不过样式很好。你觉得应该买哪款？

朋友：屏幕大的这款不错，屏幕小的这款也很好。

听了朋友的意见，你有什么感想？

案例二

团队开会讨论项目的两个完全相反的发展模式时，甲：我支持 A 模式。

乙：我赞同 B 模式。

丙：我觉得 A、B 两种模式都很好。

如果你是项目负责人，听到丙的这种意见，有什么想法？

8.1.2　肯定或否定

在很多时候，为了消除歧义，我们不需要模棱两可的表示，而是希望听到明确的肯定或否定表述，即只需要回答 yes 或 no。

使用逻辑，可以消除歧义，准确描述事物。

对于程序员来说，应该已经知道逻辑的基本是两个分支：逻辑成立、逻辑不成立。在很多场合，关于这两个分支还有很多不同表述，如真/假、是/否、Yes/No、True/False、1/0。在 C 语言系统中，对于逻辑值的表示用“非 0”（表示真）和“0”（表示假）来表示。

逻辑真	逻辑假	逻辑真	逻辑假
真	假	TRUE	FALSE
是	否	1	0
Yes	No	非 0	0

8.1.3　程序中的逻辑判断

正是由于逻辑代数（布尔代数）的创立和发展，为数字电路设计打下理论基础，才产生了今天被广泛应用到各行各业的电子计算机。

在计算机应用中，除了硬件电子线路采用逻辑电路之外，程序设计中的逻辑判断也是必不可少的。在计算机程序中使用逻辑判断功能，使设计的程序具有智能判断能力，可根据不同的条件进行分支，以执行不同的程序片断。

作为程序员的我们都知道以下公式：

程序 = 算法 + 数据

而算法又可以用以下公式来表示：

算法 = 逻辑 + 控制

由此可见，逻辑在程序设计中的重要性。

所有的程序设计语言都离不开逻辑判断语句，如 C 语言中可通过 if 语句、switch 语

句等进行逻辑判断。其他程序设计语言（如 Java、Basic 等）也提供了类似功能的语句。

逻辑判断是程序中最常用的功能。例如，根据我国相关法规的规定，年满 18 周岁的成年人才能单独进入网吧上网，因此网吧管理系统中，就需要根据上网者提供的身份证进行判断。这样就会产生两种情况：一种是已满 18 周岁，可以在网吧上网；另一种情况是未满 18 周岁，则不允许在网吧上网。这两种情况可表示为图 8-1 所示。

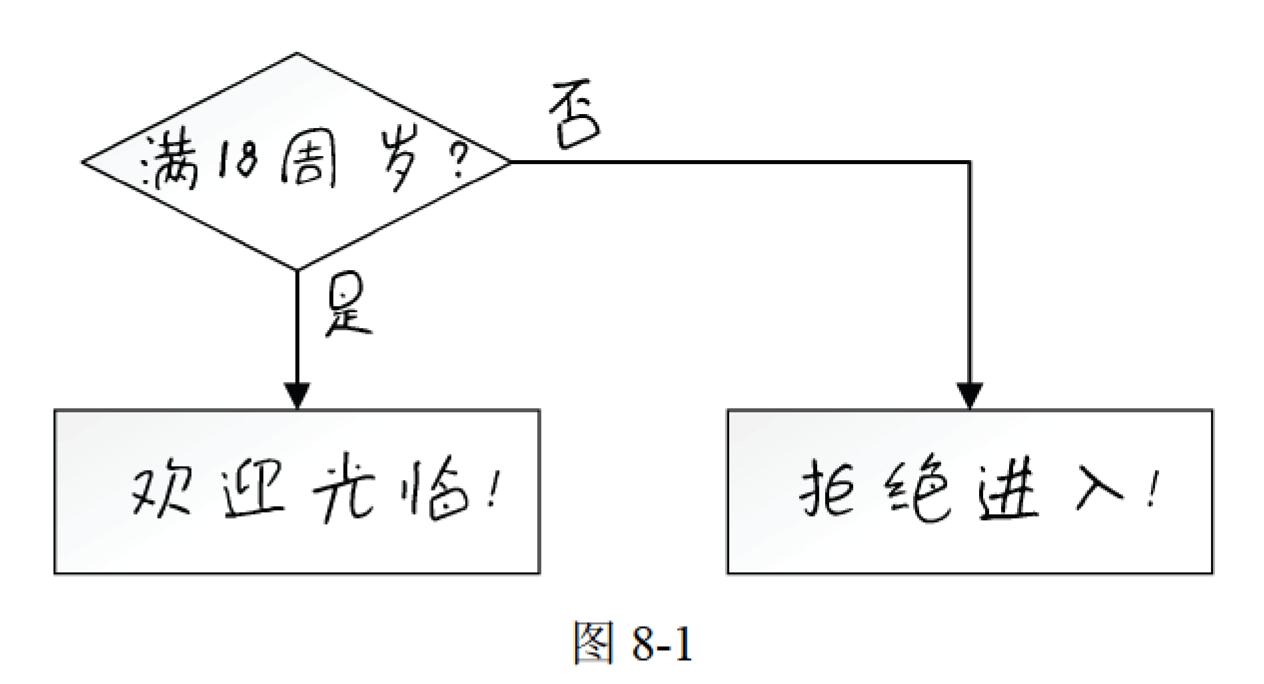


图 8-1

图 8-1 所示逻辑非常简单，可以用以下程序来实现其功能。

```
#include <stdio.h>

int main()
{
    int age;

    printf("请输入年龄:");
    scanf("%d",&age);

    if(age<18)
    {
        printf("对不起，你不能单独进入网吧！\n");
    }else{
        printf("欢迎光临！\n");
    }

    getch();
    return 0;
}
```

8.2 命题逻辑

命题逻辑是数理逻辑的基础部分，什么是命题？如何表示命题？如何构造复杂的命题？下面我们来讨论这些问题。

8.2.1 什么是命题

命题是指一个判断的语义，通俗地说，能判断真假的陈述句就称为命题。例如：

(1) 中国位于亚洲。
(2) 北京是中国的首都。
(3) 太阳从东方升起。
(4) 9是一个素数。
(5) 所有三角形内角和为360度。
(6) 2乘3的积大于10。

以上 6 个语句都是陈述句，并且每个陈述句都能判断出真假，因此，这 6 个语句都是命题。

其中（1）、（2）、（3）这 3 个语句所陈述的内容与事实相符，都是正确的，称为真命题，或称命题的值为“真”，可记为 True 或数字 1。

而（4）、（5）、（6）这 3 个语句所陈述的内容明显与事实不符，是不正确的（9 不是一个素数、三角形内角和为 180 度、2 乘 3 的积小于 10），称为假命题，或称命题的值为“假”，可记为 False 或数字 0。

接着看下面的例子：

(7) 2020年我国普通民众可坐神舟飞船上太空。
(8) 1+01等于10。
(9) 如果下雨，那么地就湿。

对于第（7）句，由于现在是 2013 年，目前只有经过特殊训练的航天员才能通过神舟飞船上太空，而 2020 年时普通民众是否能乘坐神舟飞船上太空，还是一个未知的值。但到 2020 年年底时，这句语句的值就是一个确定值了，如果到时普通民众可以坐神舟飞船上太空，则这个陈述句是正确的，为真命题，其值为 True 或数字 1；反之，则这个陈述句是不正确的，为假命题，其值为 False 或数字 0。

对于第（8）句，从十进制角度来看，其陈述的内容是不正确的；但是，从二进制角度来看，其陈述的内容又是正确的。对于这种对错不确定的陈述句，不能称为命题。

对于第（9）句所陈述的内容与事实相符，是正确的，为真命题。只是这句与前面那些陈述句有所不同，是根据一个命题推出另一个命题。

需要注意，命题必须是一个陈述句，如果为其他类型的语句，也不能称为命题。例如：

(10) 请把门关上。
(11) 中午吃什么？
(12) 今天真凉快啊！

第（10）句为祈使句、第（11）句为疑问句、第（12）句为感叹句，对于这些不是陈述句的表述，都不能称为命题。

总结一下：能够判断真假的陈述句叫做命题，正确的命题叫做真命题，错误的命题叫做假命题。命题的值要么为 True，要么为 False，不能确定为 True 或 False 的不能称为命题。

8.2.2　命题的逻辑形式

前面列举了很多命题的例子，从这些例子可以看出，命题的形式多种多样。对于这些种类繁多的命题，可使用简单的逻辑形式进行表达。命题的逻辑形式就是命题各部分之间的联系方式，许多内容上千差万别的命题，其各部分之间的联系方式可能是相同的，从而具有相同的形式。

通常可用带下标（或不带下标）的英文字母 A、B、C、……、P、Q、R、……，A_i、B_i、C_i……、P_i、Q_i、R_i 等表示命题。

对于命题：

所有的事件都有产生它的原因

可用逻辑形式表示为如下形式：

所有的 S 都是 P

即将“事件”用 S 表示，P 表示“产生它的原因”。

而对于下面的命题：

这个班所有的人都出生于 1997 年

可用逻辑形式表示为如下形式：

所有的 S 都是 P

这里用 S 表示“这个班的人”，P 表示“出生于 1997 年”。

再看一个类似形式的命题：

所有的金属都是导电的

也可用相同的逻辑形式来表示。

对于以上 3 个描述不同事物、内容有很大区别的命题，通过用逻辑形式来表示可以看出，这 3 个命题具有共同的形式。这种命题的逻辑形式相同，就是这 3 个命题所共同具有的、各部分之间的联系方式。

8.2.3　简单命题

了解命题的概念和逻辑形式之后，接下来我们来看命题的分类。根据命题是否可分解为其他命题，可将命题分为简单命题和复合命题。

简单命题是指不包含其他命题作为其组成部分的命题，即在结构上不能再分解出其他命题的命题。

简单命题一般又分为两类，一类是性质命题（直言命题），它只有一个主项和一个谓项，谓项反映的是对象的性质。例如：

(1) 金属是导电的。
(2) 有些花是红的。

另一类的是关系命题，它不限于一个主项，谓项反映的是主项之间存在的关系，例如：

(3) 武汉位于北京与长沙之间。
(4) 张三和李四是同学。

以下的例子都是简单命题：

(5) 雪花都是白色的。
(6) 今天下午5点下班。
(7) 昨天晚上下雨了。

简单命题一般难以划分前提和结论，因此简单命题的真假判断不能依靠命题逻辑推理，其真假只能依据客观事实或生活经验自行判断。例如，本节中（1）~（7）的命题中，（1）、（2）、（3）、（5）项都可以根据客观事实或生活经验判断为真命题。而第（4）项如果张三和李四是同学，则是真命题，否则为假命题。第（6）、（7）项也需要根据实际情况进行判断，如果所陈述的内容与事实相符，则是真命题，否则为假命题。

8.2.4　复合命题

复合命题是指可以分解为更简单命题的命题。而且，这些简单命题之间是通过逻辑联结词“或”、“且”、“非”、“如果……那么……”、“当且仅当”等和标点符号复合而构成的一个命题。

例如，如下命题就是复合命题。

(1) 12可以被3或4整除。
(2) 正方形的对角线互相垂直且平分。
(3) 0.8是非整数。

对于第（1）项命题，可分解为以下两个命题，并用“或”联结词联结。

(a) 12可以被3整除

或

(b) 12可以被4整除

类似地，对于第（2）项命题，可分解为：

(a) 正方形的对角线互相垂直

或

(b) 正方形的对角线互相平分

而对于第（3）项命题，很可能会将其判断为简单命题，注意其中包含了“非”联结词，这是一个复合命题，应分解为：

非（0.8是整数）

8.2.5 复合命题的联结词

命题逻辑中的联结词可归纳为 5 种：合取联结词（且）、析取联结词（或）、否定联结词（非）、条件联结词（如果……则……）、双条件联结词（当且仅当），下面分别介绍。

1．合取联结词（且）

用联结词“且”将简单命题 p 与简单命题 q 联结起来成为一个新命题，记作 $p \wedge q$，读作“p 且 q”，或“p 合取 q”。

对于复合命题 $p \wedge q$，该如何判断其真假呢？

当两个简单命题 p 和 q 都是真命题时，形成的新命题“p 且 q”就是真命题。如果两个命题 p 和 q 其中有一个是假命题，形成的新命题“p 且 q”就是假命题。

对简单命题，我们是直接以事实为依据来判定其真假。复合命题则不同，它是由联结词联结支命题而构成的，复合命题描述的是支命题之间的逻辑关联。支命题之间的逻辑关联就表现为支命题的真假对整个复合命题真假的制约关系，因此，复合命题的真假是由支命题的真假决定的。

要判断复合命题的真假，可通过真值表进行操作。将复合命题中各支命题的真假情况列在一张表中，然后根据联结词的不同，即可得到不同的真假情况。

对于用“且”联结词联结的复合命题，命题的真假由联结的两个支命题决定。例如，对于“p 且 q”这个复合命题，可制作如表 8-1 所示的真值表。

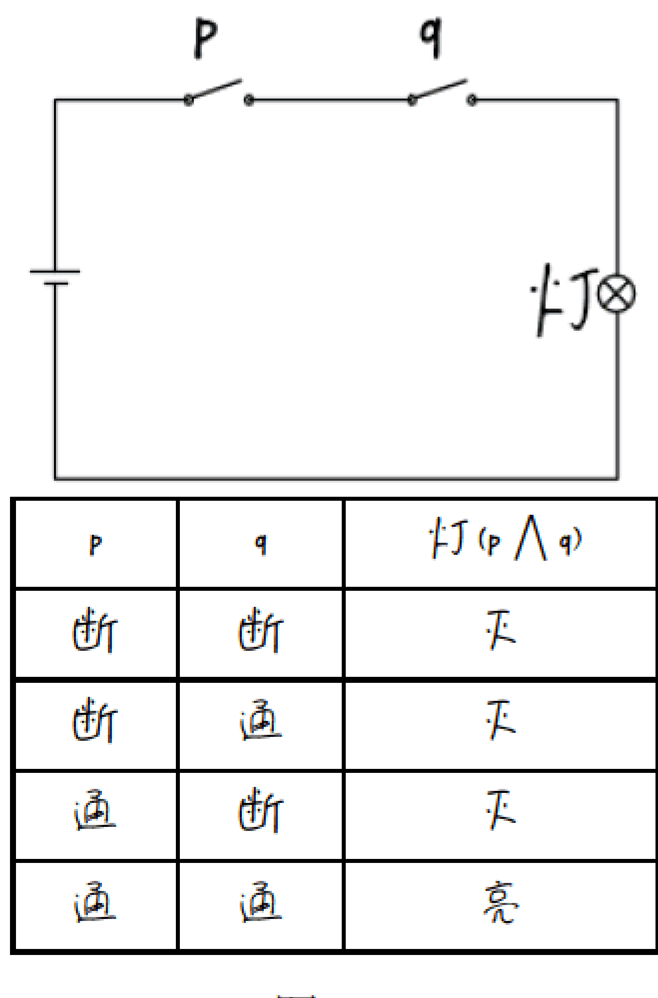
表 8-1　真值表

p	q	p 且 q
0	0	0
0	1	0
1	0	0
1	1	1

在表 8-1 中，用数字 1 表示逻辑真，数字 0 表示逻辑假。只有当 p 和 q 都是真命题时，新命题“p 且 q”才是真命题。

对于联结词“且”，还可用电路来实现，如图 8-2 所示，命题 p 和 q 各表示一个开关，灯表示复合命题“p∧q”的真假。当 p 和 q 这两个开关都接通后，复合命题“p∧q”的结果才为真（灯亮），其他情况灯都不亮。

可以看出，联结词“且”与自然语言中的“和”、“与”、“并且”、“而且”、“同时”、“即……又……”等的含义相当。



p	q	灯(p∧q)
断	断	灭
断	通	灭
通	断	灭
通	通	亮

图 8-2

2．析取联结词（或）

用联结词“或”把 p 与 q 联结起来成为一个新命题，记作 p∨q，读作“p 或 q”，或“p 析取 q”。

对于复合命题 p∨q，可按以下规则判断真假：

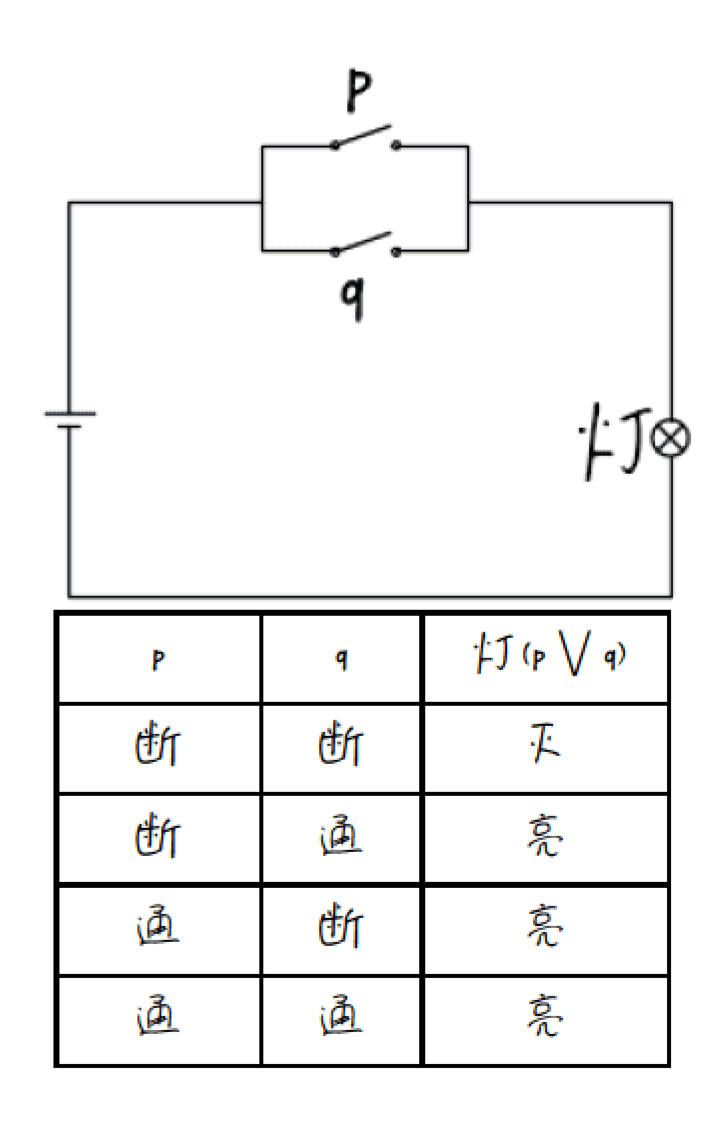
当两个命题 p 和 q 中有一个是真命题时，形成的新命题“p 或 q”就是真命题。当两个命题 p 和 q 都是假命题时，复合命题“p 或 q”就是假命题，其真值表如表 8-2 所示。

表 8-2　析取联结词真值表

p	q	p 或 q
0	0	0
0	1	1
1	0	1
1	1	1

从表 8-2 中可看到，只有当 p 和 q 都是假命题时，新命题“p 或 q”才是假命题，其他情况下，新命题“p 或 q”都是真命题。

对于联结词“或”，也可用电路来实现。如图 8-3 所示，命题 p 和 q 各表示一个开关，灯表示复合命题“p∨q”的真假。当 p 和 q 这两个开关都断开后，复合命题“p∨q”的结果才为假（灯灭），其他情况灯都亮。



p	q	灯(p∨q)
断	断	灭
断	通	亮
通	断	亮
通	通	亮

图 8-3

可以看出，联结词“或”与自然语言中的“或者”的含义相当，但是也要注意，联结词“或”与自然语言中的“或”的意义并不完全相同，自然语言中的“或”既可以表示“排斥或”，也可以表示“可兼或”，而在数理逻辑中的联结词“或”表示“可兼或”。

例如，有以下复合命题：

星期天上午我去打篮球或踢足球。

在这个复合命题中，“或”是一种“排斥或”，即“打篮球”和“踢足球”只能二选一，不能同时都为真命题。

而下面的复合命题：

参加会议的领导是董事长或总经理。

在这个复合命题中，“或”是一种“可兼或”，即“参加会议的领导是董事长”或“参加会议的领导是总经理”这两个简单命题有可能都是真命题。

3. 否定联结词（非）

对于一个命题 p，如果仅将它的结论否定，就得到一个新命题，记作¬p，读作“非 p”。

对于命题“¬p”的真假可以很简单地判定：在命题和它的非命题中，有一个且只有一个是真命题。

也就是说，在命题“非 p”中，若命题 p 为真命题，则复合命题“非 p”为假命题；反之，若 p 为假命题，则复合命题“非 p”为真命题，其真值表如表 8-3 所示。

表 8-3　否定联结词真值表

p	非 p
1	0
0	1

对于联结词“非”，也可用电路来实现。如图 8-4 所示，命题 p 表示一个开关，灯表示复合命题“¬p”的真假。当 p 开关断开后，复合命题“¬p”的结果才为真（灯亮），当 p 开关接通后，复合命题“¬p”的结果为假（灯灭）。

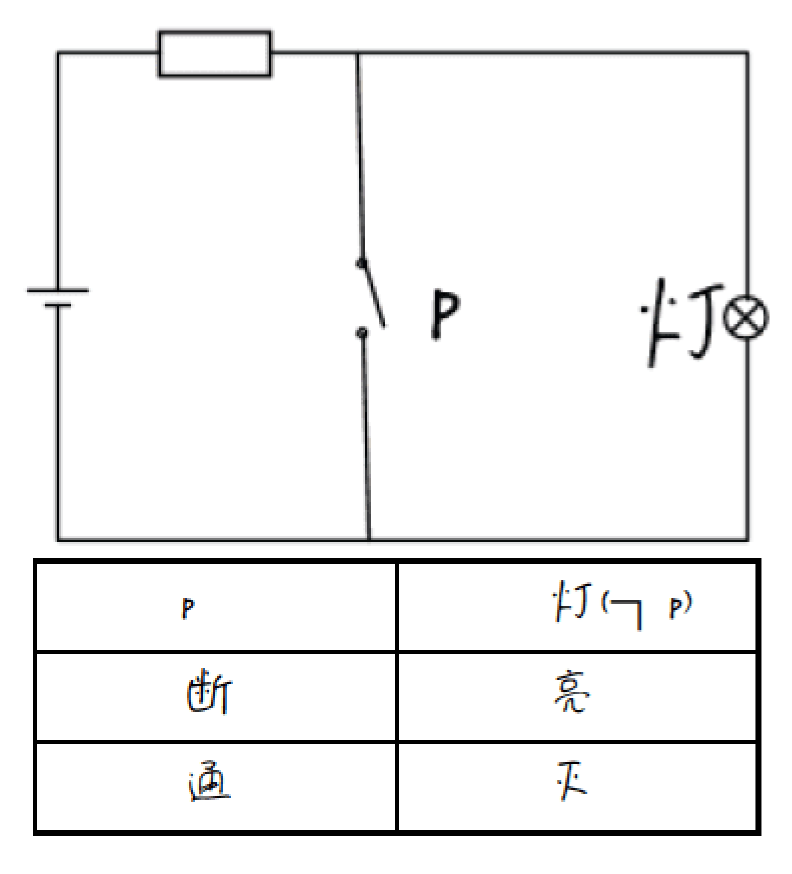


图 8-4

联结词“非”与自然语言中的“非”、“不”、“没有”等语义相当。

4. 充分条件联接词（如果……那么……）

联结词“如果……那么……”也经常使用，例如，对于命题“如果下雨，那么地就是湿的”就是用这个联结词来联结两个命题。

对于“如果 p，那么 q”这种形式的复合命题，叫做 p 蕴涵 q，记作 p→q。p 称为前件（或前提），q 称为后件（或结论）。

通过“如果……那么……”这种联结词联系的命题称为假言命题，表示事物间的条件关系如下（如图 8-5 所示）。

- 如果有事物情况 p，则必然有事物情况 q；

- 如果没有事物情况 p，未必有事物情况 q。

对于以上逻辑关系，p 就是 q 的充分而不必要的条件，简称充分条件。因此，这类假言命题也称为充分条件假言命题。

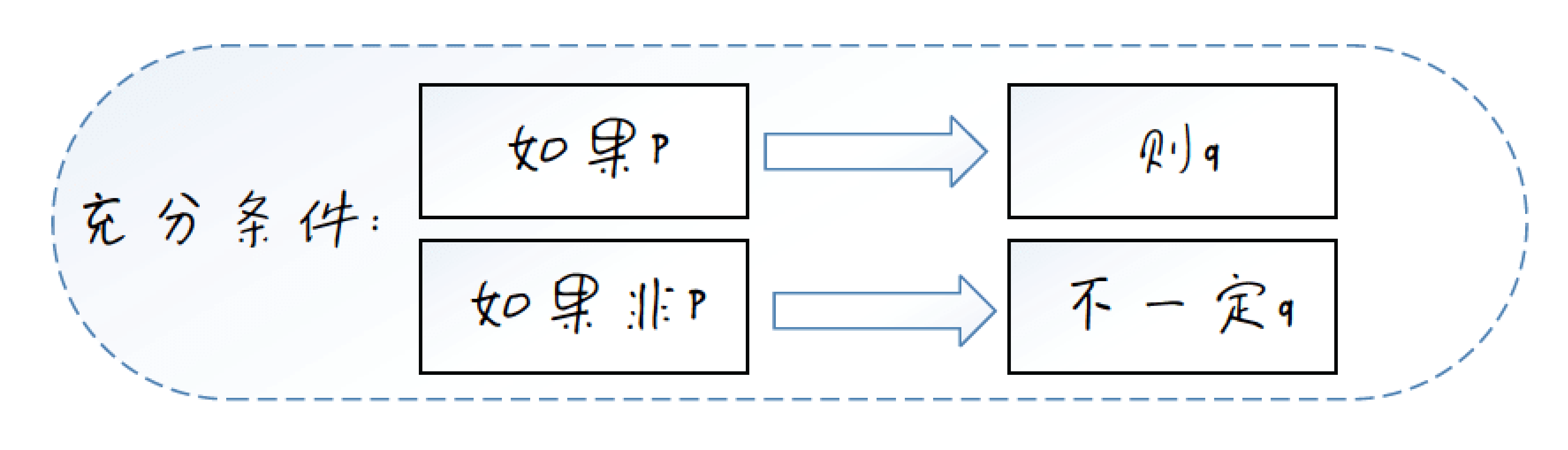


图 8-5

在自然语言中，前件为假，不管结论真假，整个语句的意义往往无法判断。但是，在充分条件假言命题逻辑中，如果前件真而后件假，则该充分条件假言命题才是假的；如果不是“前件真而后件假”，则该充分条件假言命题都是真。这种真假关系可用表 8-4 表示。

表 8-4　充分条件联结词真值表

p	q	p→q
0	0	1
0	1	1
1	0	0
1	1	1

也可这样理解：充分条件的特征是肯定前件就必须肯定后件，否定后件就必须否定前件。只否定前件是不能否定后件的，只肯定后件也不能肯定前件。

对于表 8-4 所示的真值表没有前几种联结词的真值表好理解，下面我们用一个例子来解释一下。例如，有以下复合命题：

如果下雨，那么地就会湿。

前件是“下雨”，后件是“地就会湿”。分以下 4 种情况：

（1）前件为真命题（即下雨），后件也为真命题（即地湿了），则复合命题为真命题。

（2）前件为真命题（即下雨），后件为假命题（即地没有湿），则复合命题也为假命题。

（3）前件为假命题（即不下雨），后件为真命题（即地湿了），则复合命题也为真命题。

（4）前件为假命题（即不下雨），后件为假命题（即地没有湿），则复合合题为真合题。

可以看出，只有当前件为真命题，后件为假命题时，复合命题“p→q”才是假命题。

需要注意的是，当 p 为假时，只是命题“p→q”为真，而非后件的结论 q 为真。

5．必要条件联接词（只有……才……）

联结词“只有……才……”也是经常使用的，例如，对于命题“只有年满十八周岁，才有选举权”就是用这个联结词来联结两个命题。

对于“只有 p，才 q”这种形式的复合命题，叫做由 p 逆蕴涵 q，记作 p←q。p 称为前件（或前提），q 称为后件（或结论）。

通过“只有……才……”这种联结词联系的命题也是一个假言命题，表示事物间的条件关系如下（如图 8-6 所示）。

- 如果没有事物情况 p，则必然没有事物情况 q；
- 如果有事物情况 p，而未必有事物情况 q。

对于以上逻辑关系，p 就是 q 的必要而不充分的条件，简称必要条件。因此，这类假言命题也称为必要条件假言命题。

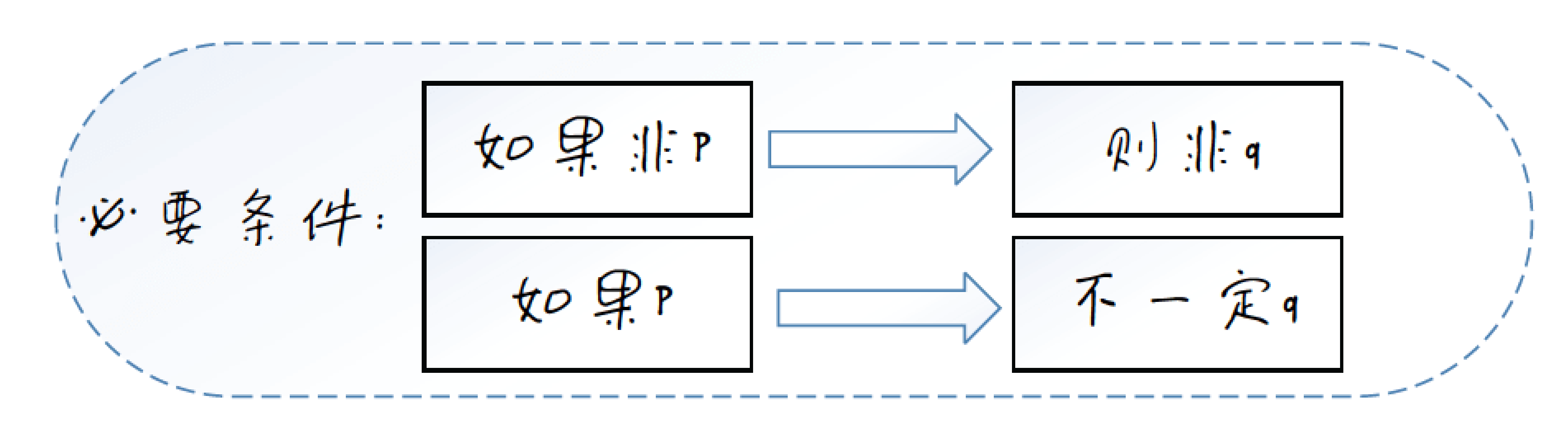


图 8-6

必要条件假言命题与其支命题（前件、后件）之间的真假关系是：如果前件假而后件真，则该必要条件假言命题才是假的；如果不是“前件假而后件真”，则该充分条件假言命题是真的。这种真假关系如表 8-5 所示真值表表示。

表 8-5　必要条件联结词真值表

p	q	p←q
0	0	1
0	1	0
1	0	1
1	1	1

6．充要条件联结词（当且仅当）

首先来看下面这个复合命题：

若冰淇淋是香草口味的，小王会吃这个冰淇淋。

这个复合命题可改成以下形式：

如果冰淇淋是香草口味的，那么小王会吃这个冰淇淋。

可以看出，这可以用条件联接词进行连接，即可用“p→q”来表示。

在命题中添加“当且仅当”后，得到如下命题：

当且仅当冰淇淋是香草口味，小王会吃这个冰淇淋。

在这个命题中，当 p（冰淇淋是香草口味）为真命题时，q（小王吃冰淇淋）为真命题；反过来，当 q（小王吃冰淇淋）为真命题时，p（冰淇淋是香草口味）必为真命题。而当 p（冰淇淋是香草口味）为假命题时，q（小王吃冰淇淋）也为假命题（即冰淇淋不是香草口味，小王不会吃）；当 q（小王吃冰淇淋）为假命题时，p（冰淇淋是香草口味）也为假命题（即小王不吃冰淇淋，则冰淇淋不为香草口味）。

如果既有 p→q，又有 q→p，就称“p 当且仅当 q”，记作 p↔q。

通过“当且仅当”这种联结词联结的命题也是一个假言命题，表示事物间的条件关系如下（如图 8-7 所示）。

- 如果有事物情况 p，则必然有事物情况 q；
- 如果没有事物情况 p，则必然没有事物情况 q。

对于以上逻辑关系，p 就是 q 的充分必要条件，简称充要条件。因此，这类假言命题也称为充分必要条件假言命题。

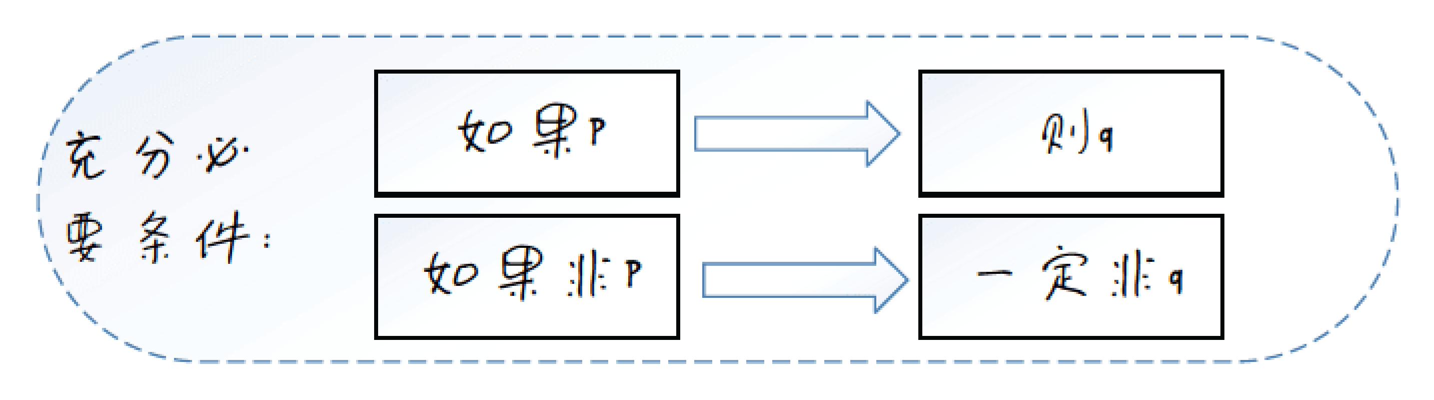


图 8-7

充分必要条件假言命题与其支命题（前件、后件）之间的真假关系是：如果前件与后件同真或同假，则该充分必要条件假言命题是真的；如果如果前件与后件不同真、不同假，则该充分必要条件假言命题是假的。这种真假关系如表 8-6 所示。

表 8-6　充要条件联结词真值表

p	q	p↔q
0	0	1
0	1	0
1	0	0
1	1	1

8.3　布尔逻辑

前面我们学习了命题逻辑，其中在复合命题中，两个命题之间可通过联接词进行联

接。对于这些联接词，我们演示了用文字和符号描述的两种方式：如“p∨q”也可描述为“p 或 q”。显然用符号描述更简洁，这种表示方式其实就是布尔逻辑的表式形式。

布尔逻辑得名于 George Boole，他在 19 世纪中叶首次定义了逻辑的代数系统，称为逻辑代数或布尔代数。

与普通代数相似，布尔代数也使用字母来表示变量。但是，与普通代数不同的是，布尔代数的运算符、运算数更简单。在布尔代数中，变量的取值只能为“1”和“0”两种，表示两种逻辑状态，即“逻辑真”或“逻辑假”。注意这里的“1”和“0”没有数值大小的含义。

布尔代数提供了 3 种基本运算，分别是逻辑乘（“与”运算），逻辑加（“或”运算）和求反（“非”运算）。

8.3.1　逻辑或

在前面学习复合命题时有一个“或”的联接词，其实就是一种“逻辑或”运算。

假设有两个简单命题 A、B，通过联接词“或”组合成复合命题，可表示为以下形式：

$$A \vee B$$

通常，逻辑变量用英文大写字母 A、B、C……表示。

从表 8-2 所示的真值表中可看出，当命题 A、B 有一个是真命题时，复合命题就是真命题，只有当 A、B 两个命题都为假命题时，复合命题才是假命题。

如下所示，当用 0 和 1 来表示逻辑假和逻辑真时，“逻辑或”运算与代数中的加法运算相似：

$$0 \vee 0 = 0 \qquad 0 + 0 = 0$$
$$0 \vee 1 = 1 \qquad 0 + 1 = 1$$
$$1 \vee 0 = 1 \quad \Rightarrow \quad 1 + 0 = 1$$
$$1 \vee 1 = 1 \qquad 1 + 1 = 1$$

因此，“逻辑或”运算又称为“逻辑加”运算。只有当两个逻辑变量的值都为 0（逻辑假）时，“逻辑或”运算的结果才为 0，其他情况下，“逻辑或”运算的结果都为 1。

“逻辑或”运算用加号将两个逻辑变量连接起来，构成如下逻辑表达式：

$$F = A + B$$

根据“逻辑或”运算的规则，不管逻辑变量 A 的值为 0 还是 1，将其与 0、1、自身相加，可得出如下结果：

$$A + 0 = A$$

$$A + 1 = 1$$

$$A + A = A$$

在程序语言中，两个逻辑变量进行“逻辑或”运算时可使用相应程序设计语言提供的“逻辑或”运算符进行操作。在 C 语言中，“逻辑或”运算符是“||”。

例如：编写一个 C 程序，判断用户输入的数据是否位于 0～10 之间。

如果要判断用户输入的数据是否位于 0～10 之间，由于数据边界有两处，则需要分两个分支进行判断。假设将用户输入的数据保存到变量 x 中，则需要用 $x>0$ 和 $x<10$ 这两个关系表达式进行判断，如图 8-8 所示。

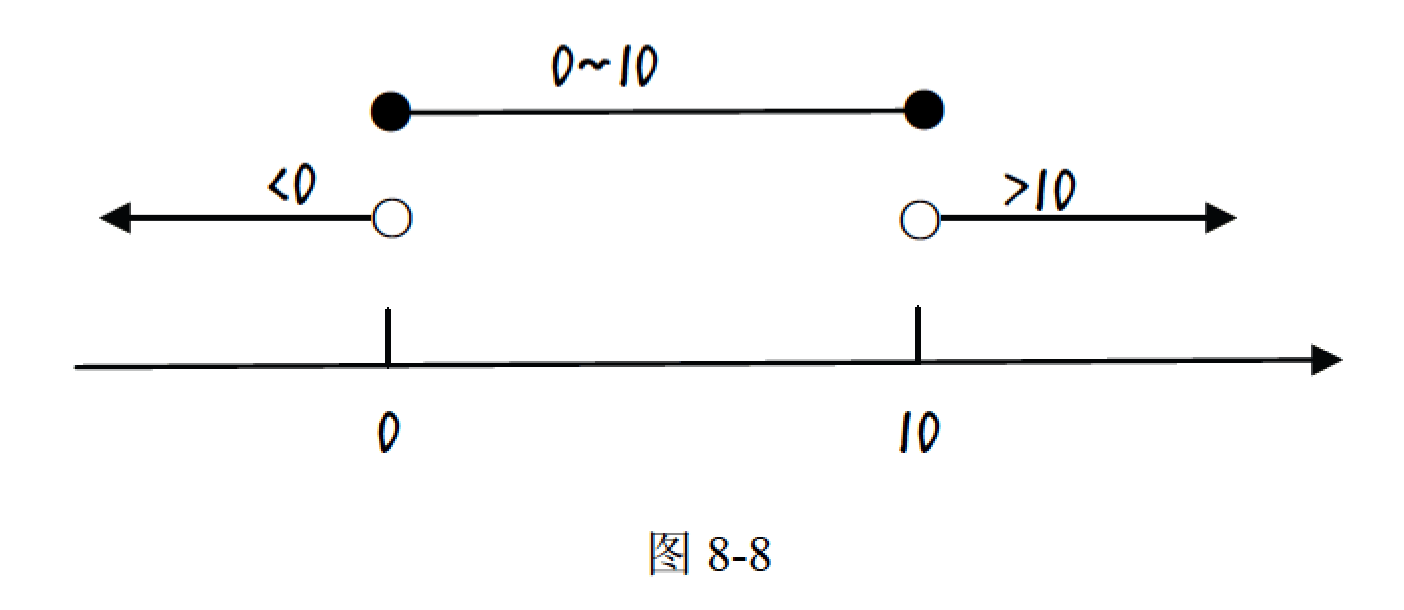


图 8-8

如图 8-8 所示，要判断数据 x 是否在 0～10 区间，需要进行多次逻辑判断：按从左向右的顺序来看，第一次需判断输入的数据是否小于 0（$x<0$），若不小于 0，说明输入的数据在图 8-8 所示的 0 坐标的右侧。接着进行第二次判断，判断 x 是否大于 10（$x>10$），再根据判断情况得出结果。

根据以上思路编写的程序如下：

```
#include <stdio.h>

int main()
{
    int x;

    printf("请输入一个整数:");
    scanf("%d",&x);

    if(x<0)
    {
        printf("输入的数不在 0~10 之间。\n");
    }else if(x>10){
        printf("输入的数不在 0~10 之间。\n");
    }else{
        printf("输入的数在 0~10 之间。\n");
    }

    getch();
    return 0;
}
```

其实，对于这种情况我们可以使用逻辑运算符将两个或多个逻辑值进行联接，然后进行判断，这样可减少代码量，使程序更简洁。

如果使用“逻辑或”运算，可将以上程序改写为以下形式：

```
#include <stdio.h>

int main()
{
    int x;

    printf("请输入一个整数:");
    scanf("%d",&x);

    if(x <0 || x>10)
    {
        printf("输入的数不在 0~10 之间。\n");
    }else{
        printf("输入的数在 0~10 之间。\n");
    }

    getch();
    return 0;
}
```

在上面的程序中，当 x>10 或者 x<0 时（两者满足其一），说明输入的数据未在 0~10 这个区间。反之，若两个条件都不满足，说明输入的数据在 0~10 这个区间。

8.3.2　逻辑与

在前面学习复合命题时有一个“且”的联接词，其实就是一种“逻辑与”运算。假设有 2 个简单命题 A、B，通过联接词“且”组合成复合命题，可表示为以下形式：

$$A \wedge B$$

从表 8-1 所示的真值表中可以看出，只有当命题 A、B 都是真命题时，复合命题才是真命题。

如下所示，当用 0 和 1 来表示逻辑假和逻辑真时，“逻辑与”运算与代数中的乘法运算相似：

$$0 \wedge 0 = 0 \qquad 0 \cdot 0 = 0$$

$$0 \wedge 1 = 0 \qquad 0 \cdot 1 = 1$$

$$1 \wedge 0 = 0 \qquad 1 \cdot 0 = 1$$

$$1 \wedge 1 = 1 \qquad 1 \cdot 1 = 1$$

因此，“逻辑与”运算又称为“逻辑乘”运算。只有当两个逻辑变量的值都为 1（逻辑真）时，“逻辑与”运算的结果才为 1。

“逻辑与”运算用乘号将两个逻辑变量连接起来，构成如下逻辑表达式：

$$F = A \cdot B$$

根据“逻辑与”运算的规则，不管逻辑变量 A 的值为 0 还是为 1，将其与 0、1、自身进行“逻辑与”运算时，可得出如下结果：

$$A \cdot 0 = 0$$

$$A \cdot 1 = A$$

$$A \cdot A = A$$

在程序语言中，两个逻辑变量进行“逻辑与”运算时可使用相应程序设计语言提供的“逻辑与”运算符进行操作。在 C 语言中，“逻辑与”运算符是“&&”。

改写上例的程序，用“逻辑与”方式判断用户输入的数据是否位于 0～10 之间。这时，我们只需要用 x>=0 和 x<=10 来判断数据是否在 0～10 之间，而不用判断数据是否在 0～10 之外。因此，程序可改为如下形式：

```
#include <stdio.h>

int main()
{
    int x;

    printf("请输入一个整数:");
    scanf("%d",&x);

    if(x>=0 && x<=10)
    {
        printf("输入的数在 0~10 之间。\n");
      }else{
        printf("输入的数不在 0~10 之间。\n");
    }

    getch();
    return 0;
}
```

8.3.3 逻辑非

逻辑非也称为逻辑反运算，就是对一个逻辑变量取相反的值，即原来为逻辑真，经过逻辑非运算后就变成逻辑假。反之，原来为逻辑假，经过逻辑非运算后就变成逻辑真了。

在前面学习复合命题联接词“非”时，用符号“¬p”表示对命题 p 进行“非 p”操

作。在逻辑表达式中也可这样书写，很多地方还可看到另外一种表示形式，即在变量上方加一条横线，为 $\overline{A}$ 的形式。

逻辑非表达式如下：

$$F = \overline{A}$$

或

$$F = \neg A$$

通常，称 A 为原变量，$\overline{A}$ 为反变量，二者共同称为互补变量。

逻辑非运算的运算规则如下（真值表参见表 8-3 所示）。

$$\overline{0} = 1$$

$$\overline{1} = 0$$

也就是说“非 0”为“1”，“非 1”为 0。根据以上规则可推出，“非（非 0）”为“非 1”（先计算括号中的“非 0”），而“非 1”为“0”，因此，“非非 0”为“0”。据此可推出“非非 A”为“A”。

$$\overline{\overline{0}} = \overline{1} = 0 \Rightarrow \overline{\overline{A}} = A$$

可以将一次“非”运算看作一次对该变量的否定，则以上推理可理解为“双重否定表示肯定”。双重否定也可表示为如下形式：

$$\neg\neg A = A$$

根据“逻辑非”运算的规则，不管逻辑变量 A 的值为 0 还是为 1，有如下运算结果：

$$\overline{\overline{A}} = A$$

$$A + \overline{A} = 1$$

$$A \cdot \overline{A} = 0$$

在程序语言中，两个逻辑变量进行“逻辑非”运算时可使用相应程序设计语言提供的“逻辑非”运算符进行操作。在 C 语言中，“逻辑非”运算符是“!”，这个运算符能够实现对表达式的条件进行取反，若表达式值为 true，则运算结果为 false；若表达式值为 false，则运算结果为 true。

例如：

!(1<2)的运算结果为 false。

!(2<1)的运算结果为 true。

8.3.4　逻辑异或

除了上面介绍的 3 个基本逻辑运算操作之外，我们经常还会用到一个称为“逻辑异

或”的运算。

逻辑异或运算用符号“ ⊕ ”连接两个逻辑变量，其表达式如下：

$$F = A \oplus B$$

对于这个特别的逻辑运算，其运算规则是什么呢？

用文字来描述运算规则，当逻辑变量 A、B 的值不同时，异或运算的结果为真；反之，当逻辑变量 A、B 的值相同时，异或运算的结果为假。其真假表如表 8-7 所示。

表 8-7　逻辑异或真值表

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

其实，“逻辑异或”运算可以转换为“逻辑与”、“逻辑或”、“逻辑非”这 3 种基本逻辑运算的组合：

$$F = A \oplus B$$

$$= \overline{A} \cdot B + A \cdot \overline{B}$$

根据表 8-7 所示的真值表，可推算出逻辑变量 A 与 0、1、自身或自身取反的运算结果如下：

$$A \oplus 0 = A$$

$$A \oplus 1 = \overline{A}$$

$$A \oplus \overline{A} = 1$$

$$A \oplus A = 0$$

也就是说，逻辑变量 A 与 0 进行异或运算，结果仍为 A；若 A 与 1 进行异或运算，结果为“非 A”；若 A 与“非 A”进行异或运算，结果为 1；A 与自身异或运算，结果为 0。

在 C 语言中，只有二进制位运算才可以使用“逻辑异或”。

8.3.5　二进制位运算

讲到逻辑运算，不得不提一下程序设计中的二进制位运算。在 C 语言中，可以对整数、布尔类型和枚举类型进行二进制位运算。C 语言中的位运算包括逻辑位运算和移位运算，我们这里学习的是逻辑运算，因此下面主要看一下二进制位的逻辑运算操作。

1．按位“与”运算

按位“与”运算符为“&”，其运算规则如下：

0 & 0 = 0

0 & 1 = 0

1 & 0 = 0

1 & 1 = 1

可以看出，其运算规则与“逻辑与”运算类似。不同的是，在运算时是将两个运算数据按位对齐，分别对各位进行运算，而不是将两个运算数据当成逻辑值来看待。

例如：计算 3&5 的结果。

要进行位运算，首先需要将两个数转换为二进制（根据计算机的位长决定位数），然后逐位对齐，再逐位进行“与”运算。

```
    0000 0011 ← 3
&   0000 0101 ← 5
---------------
    0000 0001 ← 1
```

在上面逐位进行“与”运算时，只有当对应位都为 1 时才得 1，否则结果都为 0。因此，3&5 的结果为 1。

2．按位“或”运算

按位“或”运算符为“|”，其运算规则如下：

0 | 0 = 0

0 | 1 = 1

1 | 0 = 1

1 | 1 = 1

只有当两个二进制位均为 0 时，计算结果才为 0；否则，结果均为 1。

例如：计算 3|5 的结果。

```
    0000  0011  ←  3
|   0000  0101  ←  5
----------------
    0000  0111  ←  7
```

在上面逐位进行“或”运算时，对应位中只要有 1 位为 1，就得 1，只有当对应位都为 0 时才得 0。因此，3|5 的结果为 7。

3．按位取“反”运算

按位取“反”运算符为“～”，其运算规则如下：

```
~ 0 = 1

~ 1 = 0
```

可以看出，按位取“反”运算与“逻辑反”运算类似，当二进制位为 0 时，计算结果为 1；当二进制位为 1 时，计算结果为 0。

例如：计算～5 的结果。

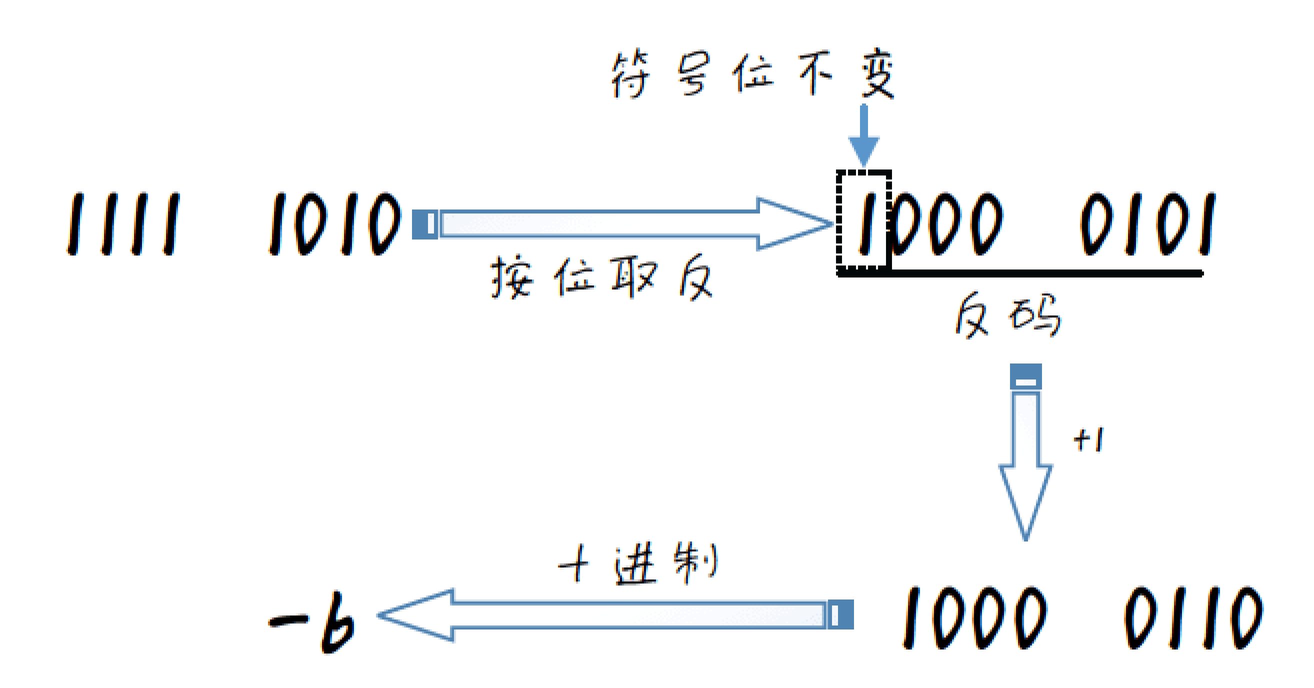
```
~   0000  0101  ←  5
----------------
    1111  1010  ←  -6
```

只看二进制位应该很好理解，取反运算将对应二进制位逐位取反（原来为 1 的变为 0，原来为 0 的变为 1）。

可是，为什么“1111 1010”表示十进制的-6 呢？

这需要从计算机中数的表示说起。简单地说，在计算机中数据的机器码采用补码表示，其中最高位为符号位，当为 1 时表示这个数是负数，为 0 时表示这个数为正数。对于正数，其补码与原码相同；对于负数，其补码是原码按位取反再加上 1。

上面取反计算的结果中，最高位取反后变为 1，表示这个数为负数。因此，要知道其原码还需要进行转换。将补码再次按位取反再加上 1 就可以得原码（符号位不变），因此，可按以下方式转换得到原码：



可见，在计算机中，对数据进行取“反”的操作很多，仅在补码的转换中就会经常用到。

4．按位“异或”运算

按位“异或”运算符为“^”，其运算规则如下：

$$0 \wedge 0 = 0$$

$$0 \wedge 1 = 1$$

$$1 \wedge 0 = 1$$

$$1 \wedge 1 = 0$$

当两个二进制位相异（不相同）时，计算结果为 1；当两个二进制位相同时，计算结果为 0。

例如：计算 3^5 的结果。

```
   0000 0011 ← 3
^  0000 0101 ← 5
-----------
   0000 0110 ← 6
```

在上面逐位进行“异或”运算时，对应位如果相同得 0，对应位如果不相同得 1。因此，3^5 的结果为 6。

8.4　考虑到各种可能了吗

我们经常可以在媒体中看到有一些招聘、招生等信息，这些信息中通常都会对应聘者（或应考者）的资格做一些限制，并且这些限制信息一般描述得比较详细。可是，如果我们仔细对这些信息进行分析，可能会发现有的单位发布的资料限制信息存在漏洞(或者条件有重叠)，让人读起来感觉不够清晰、明了。

这时，我们可以运用命题逻辑来对这些资格限制条件进行分析。将发布的资格限制条件看作为一个复合命题，在进行分析时，通常需要将复合命题分解成简单命题，以方便用逻辑表达式来表示。对分解出的简单命题进行分析，可以查看复合命题的描述有没有重复和遗漏之处。

8.4.1　逻辑重叠的实例

首先来看一个逻辑重叠的实例。

某市教育局为了解决外来务工人员子女上学问题，每年都要举办外地生招生考试，

今年举行该考试之前，教育局公布了参加此次考试考生的资格限制条件。该教育局官网发布公告表述为：

符合以下条件之一的学生可以参加本次网上报名。

（1）非本市中心城区户籍，且未在本市中心城区参加高中阶段教育学校统一招生考试报名的初三毕业生。

（2）本市中心城区户籍，且未在本市中心城区参加高中阶段教育学校统一招生考试报名的初三毕业生。

通过对以上两条限制条件进行分析，可分解出两个简单命题：

A. 本市中心城区户籍

B. 参加本市中心城区高中招生考试

定义了这两个简单命题之后，就可以定义每个条件的逻辑表达式了。

第一个条件的命题公式为：

$$\overline{A} \wedge \overline{B}$$

第二个条件的命题公式为：

$$A \wedge \overline{B}$$

符合以上两个条件之一，就有报名资格。也就是说，这两个条件之间可用联接词“或”进行联接，于是可得到如下命题公式：

$$(\overline{A} \wedge \overline{B}) \vee (A \wedge \overline{B})$$

根据逻辑运算的相关规则，将以上命题公式转换为如下逻辑表达式：

$$(\overline{A} \cdot \overline{B}) + (A \cdot \overline{B})$$

对于以上逻辑表达式，可进行化简运算，具体计算过程如下：

$$(\overline{A} \cdot \overline{B}) + (A \cdot \overline{B})$$

$$= (\overline{A} + A)\overline{B}$$

$$= \overline{B}$$

可以看出，经过化简后，看起来很复杂的一个逻辑表达式最后只剩一项，即“非 B”。也就是说，只需要满足“非（参加本市中心城区高中招生考试）”，即只要没有参加本市中心城区高中招生考试，就满足报名条件。而该市教育局发布的公告中的条件就有了重叠，而这种重叠让人看起来觉得条件较多，需要分门别类去核对自己是否满足报名条件。其实质就只有一个条件：“未在本市中心城区参加高中阶段教育学校统一招生考试报名的初

三毕业生”。

提示，本章后面还要介绍一种用卡诺图化简逻辑表达式的方法。

对于以上逻辑重叠的实例，不会影响考生的判断，不同类别的考生能够根据条件作出正确的判断，只是稍显啰嗦而已。

我们还可能会遇到另外一种逻辑重叠，且重叠部分自相矛盾，使得规则不符合逻辑的情况。看下面的例子：

某超市开展促销活动，顾客单张购物小票达到一定金额后将送购物券，具体的送券条件如下：

500~1000 元，送 50 元购物券 1 张

1000~2000 元，送 50 元购物券 2 张

2000 元以上，送 50 元购物券 3 张

列一个表格，看看不同购物金额可领取的购物券金额，如表 8-8 所示。

表 8-8　不同购物金额对应的购物券

	购 物 金 额	购　物　券
	300 元	无
	500 元	50 元 1 张
→	1000 元	50 元 1 张
→	1000 元	50 元 2 张
	1500 元	50 元 2 张
→	2000 元	50 元 2 张
→	2000 元	50 元 3 张
	2100 元	50 元 3 张

可以看出，表 8-8 中标了箭头的行都存在重叠，按送券规则第 1 条，购物 1000 元只能领取 50 元购物券 1 张；但是按第 2 条规则，1000 元又可领取 50 元购物券 2 张。类似地，2000 元时也存在重叠的问题。

8.4.2　逻辑遗漏的实例

在定义逻辑规则时有可能出现重叠，当然也有可能出现遗漏。仍然以超市促销政策为例，假设定义以下规则：

不足 500 元，送矿泉水一瓶

超过 500 元，送速溶咖啡一盒

可以看出，以上规则只定义了购物金额不足 500 元（<500）和超过 500 元（>500）的情况，但是，遗漏了购物金额正好是 500 元的情况。这样，就使得定义的规则不恰当了。

可以将以上规则修改为以下形式：

不足500元，送矿泉水一瓶

500元以上，送速溶咖啡一盒

将“超过 500 元”修改为“500 元以上”，就将所有情况都包含了。

通常，用“xx 以上”或“xx 以下”，都表示包含“xx”这个值。因此，如果将规则写成以下形式，又会出现重叠的情况（两个规则都包含 500 元）：

500元以下，送矿泉水一瓶

500元以上，送速溶咖啡一盒

8.4.3 用数轴确定边界

通过前面的例子可看出，在定义规则时，如果稍有疏忽就会导致定义的规则出现重叠或遗漏。并且，最容易出现重叠或遗漏的值通常是我们定义规则的边界值。例如，在前面超市促销的例子中，以 500 元为边界，容易出现漏掉 500 元的情况，也可能出现两个规则中都包含 500 元的这种情况。

那么，有没有什么好的方法可以辅助我们进行边界的界定呢？

在初中数学中，我们曾学过用数轴理解不等式的解集。其实，我们通常定义的规则也是一种不等式，因此，也可考虑使用数轴来辅助确定边界。

例如，将以下规则绘制到数轴中，如图 8-9 所示。

不足500元，送矿泉水一瓶

超过500元，送速溶咖啡一盒

超过500

不足500

0

500

图 8-9

在数轴图形中，空心圆表示在规则中不包含该点的值。如图 8-9 所示，则表示购物金额大于 0 且小于 500 的为“不足 500”，而“超过 500”表示购物金额大于 500。从图 8-9 中可直观的看到，500 元这个点没有包含到任何规则中。这样，通过数轴图形就可以快速发现被遗漏的点。

而对于以下规则，可用图 8-10 所示的数轴来表示。

500元以下，送矿泉水一瓶

500元以上，送速溶咖啡一盒

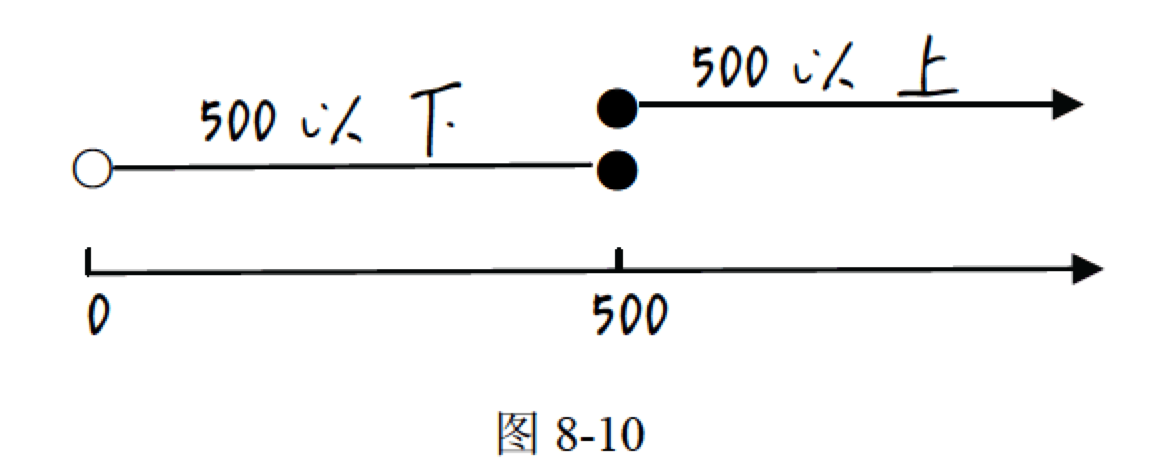


图 8-10

在图 8-10 中，实心圆表示在规则中包含该点的数据。从图 8-10 中可看出，“500 以下”中包含 500，而“500 以上”中也包含 500。因此，可直观地发现 500 元这个点重叠了。

而以下规则可用图 8-11 所示的数轴图来表示。



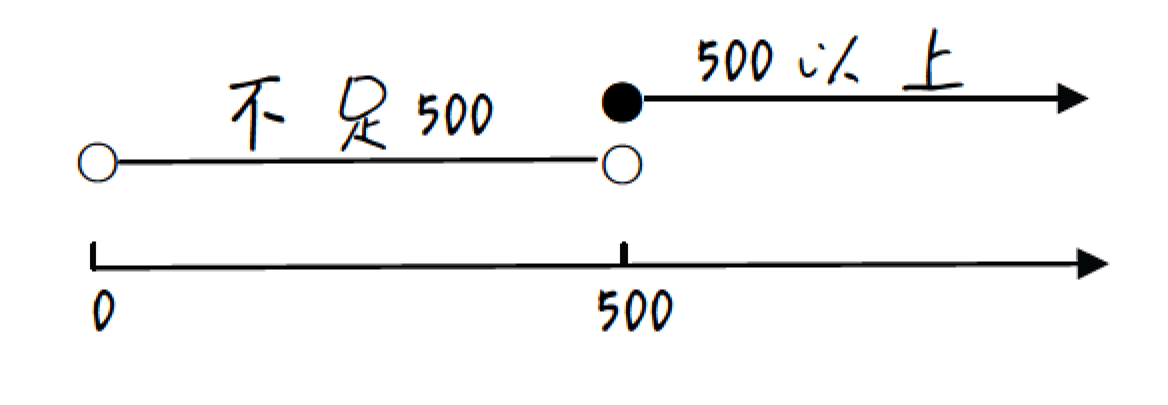


图 8-11

在图 8-11 中，“500 以上”规则中 500 处是实心圆，表示在该规则中包含 500 这个点，而“不足 500”规则中 500 处是空心圆，表示在该规则中不包含 500 这个点。即 500 这个点在两个规则中被（且只被）一个规则包含，没有出现遗漏或重叠，因此这两个规则是合符逻辑的。

8.5　用卡诺图简化逻辑函数

在 8.4 节的例子中，我们演示了通过逻辑运算简化逻辑函数的方法。其实，还可以通过图形方式，以直观、简洁的方式简化逻辑函数。本节我们就来学习这种通过卡诺图简化逻辑函数的方法。

8.5.1　什么是卡诺图

卡诺图是逻辑函数的一种图形表示。一个逻辑函数的卡诺图就是将此函数所有命题的真假组合以二维表的形式表示的图形。

根据以上定义我们可以知道，卡诺图是一种平面方格图。另外，在卡诺图中要表示每个命题的真假情况。也就是说，每一个命题都要表现两种情况，分别用两个方格来表

示，每个小方格代表一个最小项（即一个命题的真或假）。因此，卡诺图也称为最小方格图。

只看概念比较抽象，来看一个实际例子。仍以前面的外地生招生考试为例，将该例子用卡诺图来表示。该例可分解出两个简单命题：

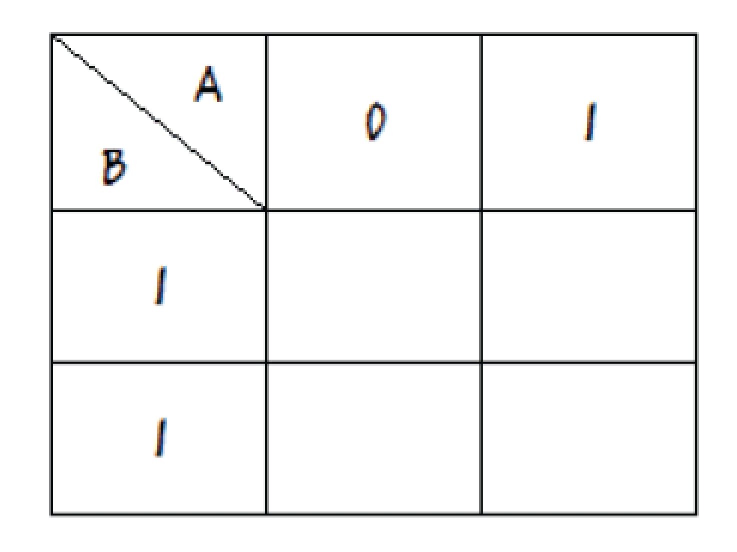
A. 本市中心城区户籍

B. 参加本市中心城区高中招生考试

下面就根据设置的规则，将这两个简单命题所有真假组合绘制成一个卡诺图，具体步骤如下。

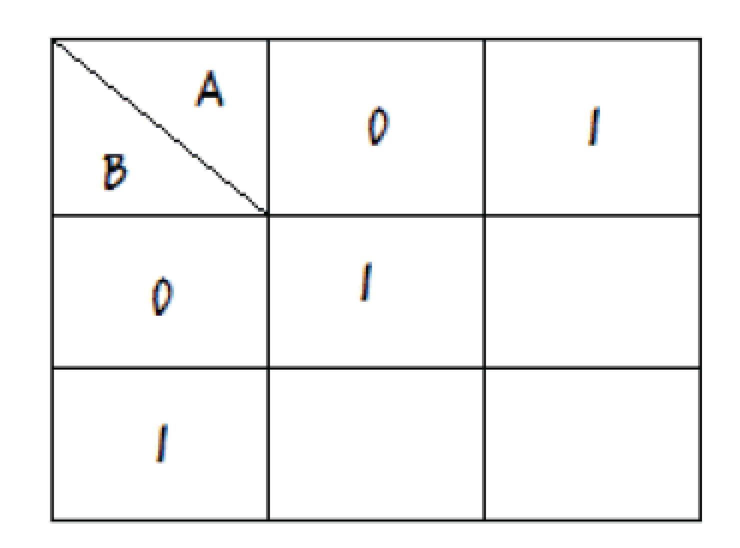
（1）绘制如图 8-12 所示的二维表格，在这个表格中，命题 A 的两个值（真、假）放置在第 1 行右侧的两列中（其中 0 表示假、1 表示真）；类似地，命题 B 的两个值（真、假）放置在第 1 列下面的两行中。

（2）根据命题描述的情况，在图 8-12 对应方格中标记 1。针对报名规则 1（“非本市中心城区户籍，且未在本市中心城区参加高中阶段教育学校统一招生考试报名的初三毕业生”）可知道，这句话的两个命题为“非 A，非 B”），则在 A 为 0、B 为 0 对应的方格中标记 1，得到如图 8-13 所示的卡诺图。



B \ A	0	1
1		
1		

图 8-12



B \ A	0	1
0	1	
1		

图 8-13

（3）继续在卡诺图对应方格中进行标记。针对报名规则 2（“本市中心城区户籍，且未在本市中心城区参加高中阶段教育学校统一招生考试报名的初三毕业生”）可知道，这句话的两个命题为“A，非 B”，则标记后的卡诺图如图 8-14 所示。

B \ A	0	1
0	1	1
1		

图 8-14

图 8-14 所示的卡诺图是根据以下逻辑函数绘制的：

$$(\overline{A}\cdot\overline{B})+(A\cdot\overline{B})$$

可以看到，这个逻辑函数中只有两个逻辑变量，如果有多个逻辑变量，又该怎么绘制卡诺图呢？

8.5.2　三变量卡诺图

卡诺图是一个二维图表，如果有多个变量，则需要在行/列中同时显示多个变量，这样，行/列中的方格将不再是两行两列，而是根据变量数量的两倍来绘制。

例如，要绘制以下逻辑函数的卡诺图：

$$F = A\overline{B}\overline{C} + \overline{A}\overline{B}C + ABC$$

可以看出，这个逻辑函数有 3 个逻辑变量，该怎么绘制 3 个变量的卡诺图呢？

（1）绘制如图 8-15 所示的二维表格，在这个表格中，用 4 列来表示逻辑变量 B、C 的真假情况（这应该很好理解，逻辑变量 B、C 共有 4 种组合形式）。而逻辑变量 A 位于行中，仍然用两行来表示真假两种情况。

A \ BC	00	01	11	10
0				
1				

图 8-15

注意，在图 8-15 所示表格中，列中数据的变化，“01”右侧是“11”，很多人根据二进制变化规则会觉得“01”右侧应该是“10”，然后“10”的右侧才是“11”。这是卡诺图相邻项的要求，学习了后面卡诺图化简后就能明白为什么要这样做了。

（2）接下来就是根据逻辑函数进行填值了。这时，不再像二变量（8.5.1 节例子）的卡诺图可以按行、列对应来查找了，在三变量即本节例中时，情况变得有点复杂了。不过，我们可以用另一种方式来标记方格，即将逻辑函数中各变量转变为 0、1 两个数，然后在卡诺图中找到对应位置。转变时，原变量取值为 1，反变量取值为 0。

$$F = A\overline{B}\overline{C} + \overline{A}\overline{B}C + ABC$$

$A\overline{B}\overline{C}$ ⇕ 100　　$\overline{A}\overline{B}C$ ⇕ 001　　ABC ⇕ 111

（3）根据转变后的二进制数据，找到对应的方格，如图 8-16 所示。

A \ BC	0 0	0 1	1 1	1 0
0		$\bar{A}\bar{B}C$		
1	$A\bar{B}\bar{C}$		ABC	

图 8-16

（4）在找到的对应方格中标记 1 即可，如图 8-17 所示。

A \ BC	0 0	0 1	1 1	1 0
0		1		
1	1		1	

图 8-17

对于三变量卡诺图，在前面例子中采用的是在行方向上排列 1 个变量，列方向上排列 2 个变量。其实也可以反过来，在行方向上排列 2 个变量，列方向上排列 1 个变量，如图 8-18 所示。

由此可见，卡诺图的表示方法并不是唯一的，可以有很多种，例如，还可将 A 放在列，而将 BC 放在行中等。

AB \ C	0	1
0 0		$\bar{A}\bar{B}C$
0 1		
1 1		ABC
1 0	$A\bar{B}\bar{C}$	

图 8-18

8.5.3 四变量卡诺图

继续增加难度，我们看看四变量的卡诺图该怎么绘制。

有如下逻辑函数，要求将其绘制在卡诺图中。

$$F = \bar{A}D + AB + B\bar{C}D$$

对于这个逻辑函数，我们发现有 4 个逻辑变量 A、B、C、D，并且在每一个逻辑式中都只有 2 个或 3 个逻辑变量组成，这在卡诺图中该怎么表示呢？

（1）绘制如图 8-19 所示的四变量卡诺图。

AB \ CD	00	01	11	10
00				
01				
11				
10				

图 8-19

（2）第一项 $\bar{A}D$ 只包含了 2 个逻辑变量，变量 B、C 都没有出现，对于这种情况，应将 A=0，D=1 的方格都填上 1，如图 8-20 所示。

AB \ CD	00	01	11	10
00		1	1	
01		1	1	
11				
10				

图 8-20

（3）用类似的方法，可将逻辑函数中其他两项也对应标记到卡诺图中，如图 8-21 所示。其中 AB 对应同时满足 A=1，B=1 的方格；而 $B\bar{C}D$ 对应同时满足 B=1，C=0，D=1 的方格。

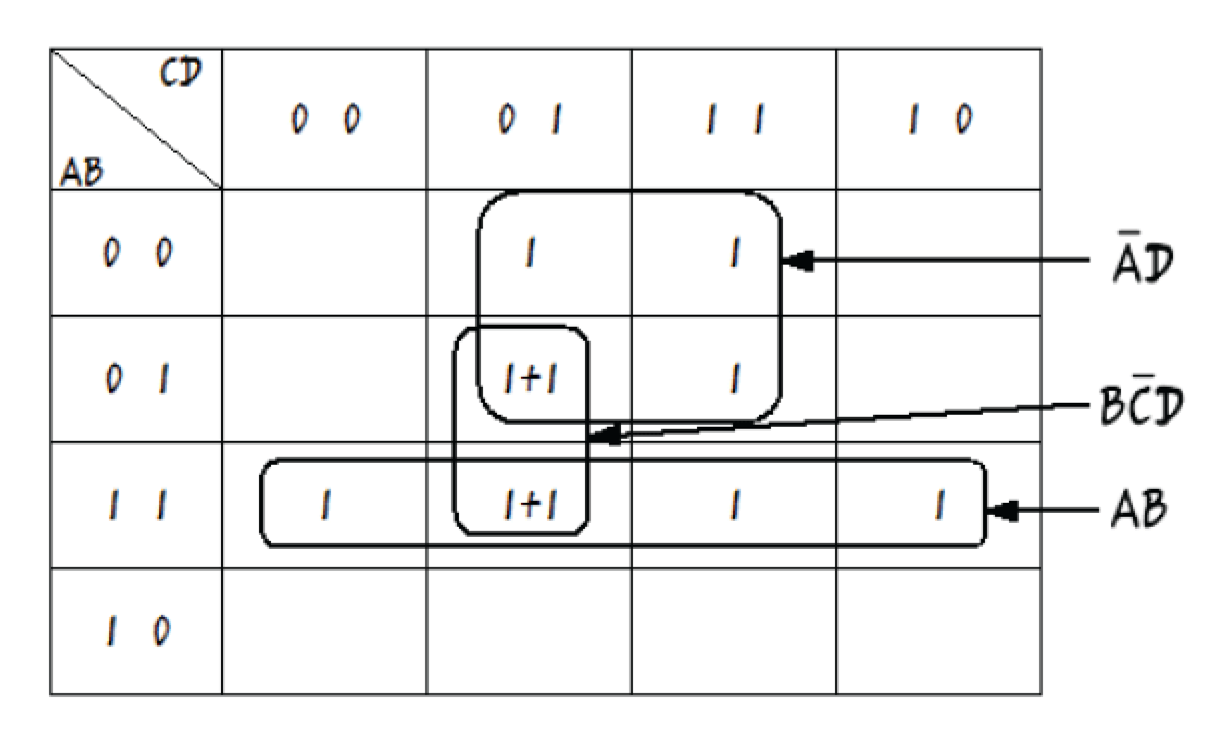


图 8-21

从理论上来说，卡诺图可以表示任意多个逻辑变量的情况，不过实际应用中，一般用于四变量及以下逻辑变量的表示和化简操作，因此，更多变量的卡诺图就不介绍了。

8.5.4 卡诺图化简

使用卡诺图的目的就是化简逻辑函数，学习绘制卡诺图也是为化简打下基础。首先，我们来看一下卡诺图的化简规则。

化简规律：两个相邻最小项有一个变量相异，相加可以消去这一个变量，化简结果为相同变量的与。

这句话是什么意思呢？来看一个如图 8-22 所示的卡诺图。

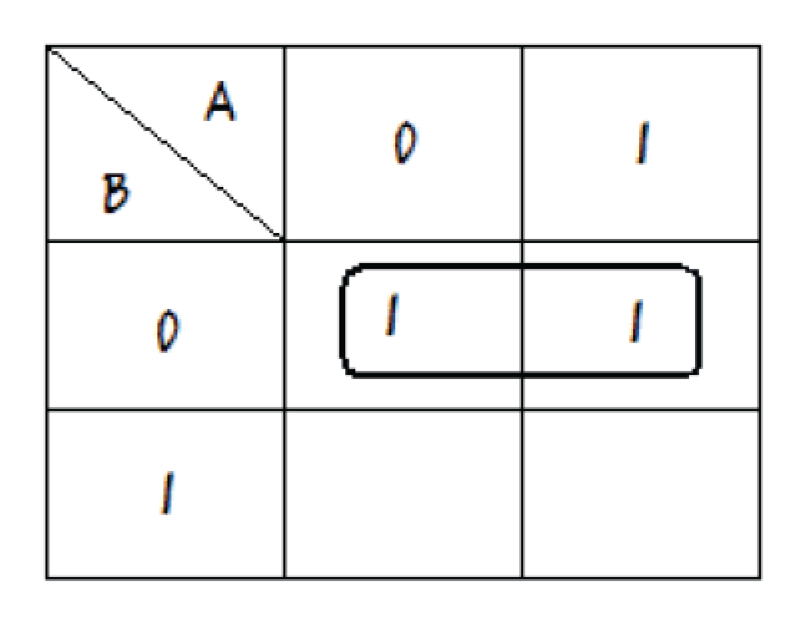


图 8-22

在该图中圈起来的两个相邻方格可用以下逻辑函数来表示：

$$\overline{A}\,\overline{B} + A\overline{B}$$

可以看出，在卡诺图中两个相邻方格中，只有一个变量取值不同（在图 8-22 中是变量 A 的取值不同），而其余的取值都相同（图 8-22 中变量 B 的取值相同）。所以，合并相邻方格可利用公式：

$$\overline{A} + A = 1$$

从而使

$$\overline{A}\,\overline{B} + A\overline{B}$$

$$=(\overline{A} + A)\overline{B}$$

$$=\overline{B}$$

这样，就可消去一个变量，从而使逻辑函数得到简化。

使用类似的方式可进行 4、8 个相邻项的合并，可发现卡诺图中相邻项合并的规律：

- ❑ 合并相邻最小项，可消去变量。

- ❑ 合并 2 个最小项，可消去 1 个变量。
- ❑ 合并 4 个最小项，可消去 2 个变量。
- ❑ 合并 8 个最小项，可消去 3 个变量。
- ❑ 合并 2^n 个最小项，可消去 n 个变量。

来看一个例子。用卡诺图化简以下逻辑函数：

$$AB\overline{C}D + ABCD + A\overline{B}\,\overline{C}D + A\overline{B}CD$$

首先，根据逻辑函数绘制出如图 8-23 所示的卡诺图。

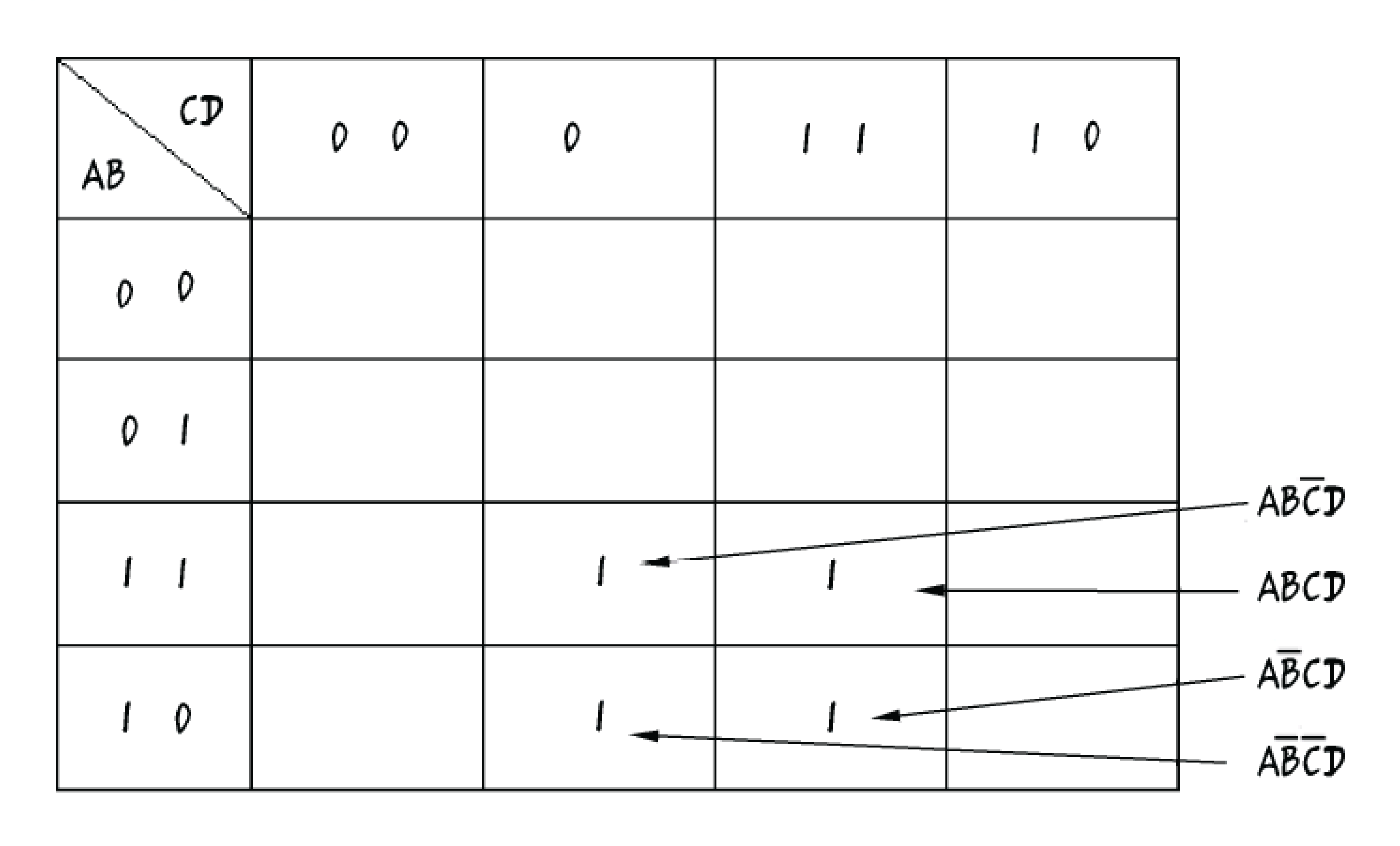


图 8-23

可以看出，在图 8-23 所示的卡诺图中，有 4 个相邻方格标记为 1，可对其进行简化，4 个相邻方格可简化 2 个逻辑变量。

那么，从图 8-23 所示卡诺图中简化后的逻辑函数是什么？该怎么从图中得出简化后的逻辑函数呢？

对图 8-23 所示卡诺图，未标记 1 的位置不用关心。对于标记为 1 方格对应的各变量进行查看，在本例中将 4 个相邻方格圈出来；然后将相邻方格中变量值相异部分圈出来，得到图 8-24 所示内容。

CD / AB	0 0	0 1	1 1	1 0
0 0				
0 1				
1 1		1	1	
1 0		1	1	

图 8-24

从图 8-24 中可以看出，AB 部分中变量 A 一直为 1，而变量 B 是相异的，因此可消去变量 B；再看 CD 部分，变量 C 是相异的，而变量 D 一直为 1。因此，最后得到的应该是变量 A 为 1，变量 D 为 1，即：

$$AB\overline{C}D + ABCD + A\overline{B}\overline{C}D + A\overline{B}CD$$

$$=AD$$

从以上简化过程可以看出，在用卡诺图化简逻辑函数时通常按以下步骤进行。

（1）按 2、4、8、2^n 个数量的规则来圈取值为 1 的方格。

（2）将上一步圈选各方格中互补的因子消去（即逻辑变量值相异的），保留相同的因子，相同因子取值为 1 表示用原变量表示，相同因子取值为 0 表示用反变量表示，将保留的因子按“与”的关系连接（即连续写在一起）。

（3）重复第（1）、（2）步，继续按 2^n 个方格数量的方式圈选其他相邻方格，消去互补因子，保留相同的因，写出“与”项表达式。再将该“与”项表达式与前面 1～2 步写出的“与”项表达式进行“或”连接（即用加号连起来）。

（4）重复第（3）步，最后即可得到化简后的逻辑函数。

8.5.5 卡诺图中的相邻

可以看出，在卡诺图中“相邻”是一个很重要的概念。在卡诺图中，应将表格看作为一个循环的状态，这样才能理解相邻的概念。例如，如图 8-25 所示，其中标识①的框内圈选了 8 个方格，可以很容易看出这 8 个方格是相邻的；可是，标识②的框（注意左、右都只有半边）也是相邻的，这种情况可能不好理解。

AB \ CD	0 0	0 1	1 1	1 0
0 0	1	1	1	1
0 1	1	1	1	1
1 1	1			1
1 0	1			1

图 8-25

我们可以换一种方式来理解，将第 1 列复制一份到表格右侧，如图 8-26 所示，怎么样？这样看标识②的框中 4 个方格，可以很容易看出是相邻的了吧。也就是说，第 1 列与最后 1 列之间是相邻的关系。类似地，第 1 行与最后 1 行之间也是相邻的关系。

AB \ CD	0 0	0 1	1 1	1 0	0 0
0 0	1	1	1	1	1
0 1	1	1	1	1	1
1 1	1			1	1
1 0	1			1	1

图 8-26

最后，我们再来看一下前面提到过的一个问题，为什么“01”列的右侧是“11”而不是“10”？其实，这也是从相邻方格合并简化方面来考虑的。仔细观察图 8-26 所示的卡诺图，无论是列还是行，相邻行或相邻列中的两个逻辑变量之间始终都只有一个发生变化。如“01”列右侧变“11”，只有第 1 个（左侧）逻辑变量从 0 变为 1，而第 2 个（右侧）逻辑变量未变化。这样，在化简时才能方便地确定一个相异逻辑变量，一个相同逻辑变量。

反之，如果在“01”右侧为“10”，可以看到第 1 个（左侧）逻辑变量从 0 变为 1，第 2 个（右侧）逻辑变量从 1 变为 0，两个逻辑变量都在变化，就没办法进行化简了。

第 9 章　推理——逻辑的应用

所谓推理，是指由一个或几个已知的判断（前提），推导出一个未知结论的思维过程。推理的作用是从已知的知识得到未知的知识，特别是可以得到不可能通过感觉、经验掌握的未知知识。推理主要有演绎推理和归纳推理。演绎推理是从一般规律出发，运用逻辑证明或数学运算，得出特殊事实应遵循的规律，即从一般到特殊。归纳推理就是从许多个别的事物中概括出一般性概念、原则或结论，即从特殊到一般。

本章我们先介绍一些演绎推理和归纳推理的基础，然后通过一个推理案演示推理在实际生活中的应用。

9.1　演 绎 推 理

所谓演绎推理，就是从一般性的前提出发，通过推导即“演绎”，得出具体陈述或个别结论的过程。

9.1.1　认识演绎推理点

从演绎推理的定义可以看出，演绎推理是从一般到特殊的推理。

演绎推理是前提和结论之间具有必然联系的推理，是前提与结论之间具有充分条件或充分必要条件联系的必然性推理。

演绎推理的逻辑形式对于理性的重要意义在于，它对人的思维保持严密性、一贯性有着不可替代的校正作用。这是因为演绎推理保证推理有效的根据并不在于它的内容，而在于它的形式。演绎推理最典型、最重要的应用，通常存在于逻辑和数学证明中。

演绎推理的主要作用如下。

- 检验假设和理论：演绎法对假说做出推论，同时利用观察和实验来检验假设。
- 逻辑论证的工具：为科学知识的合理性提供逻辑证明。
- 作出科学预见的手段：把一个原理运用到具体场合，作出正确推理。

演绎推理是一种必然性推理，推理的前提是一般，推出的结论是个别，一般中概括了个别。

事物有共性，必然蕴藏着个别，所以“一般”中必然能够推演出“个别”，而推演出来的结论是否正确，取决于大前提是否正确，推理是否合乎逻辑。

演绎法也有其局限，推理结论的可靠性受前提（归纳的结论）的制约，而前提是否

正确在演绎范围内是无法解决的。

演绎推理常见的形式有三段论、选言推理、假言推理、关系推理等，这些分类中又可进一步细分，如图 9-1 所示。

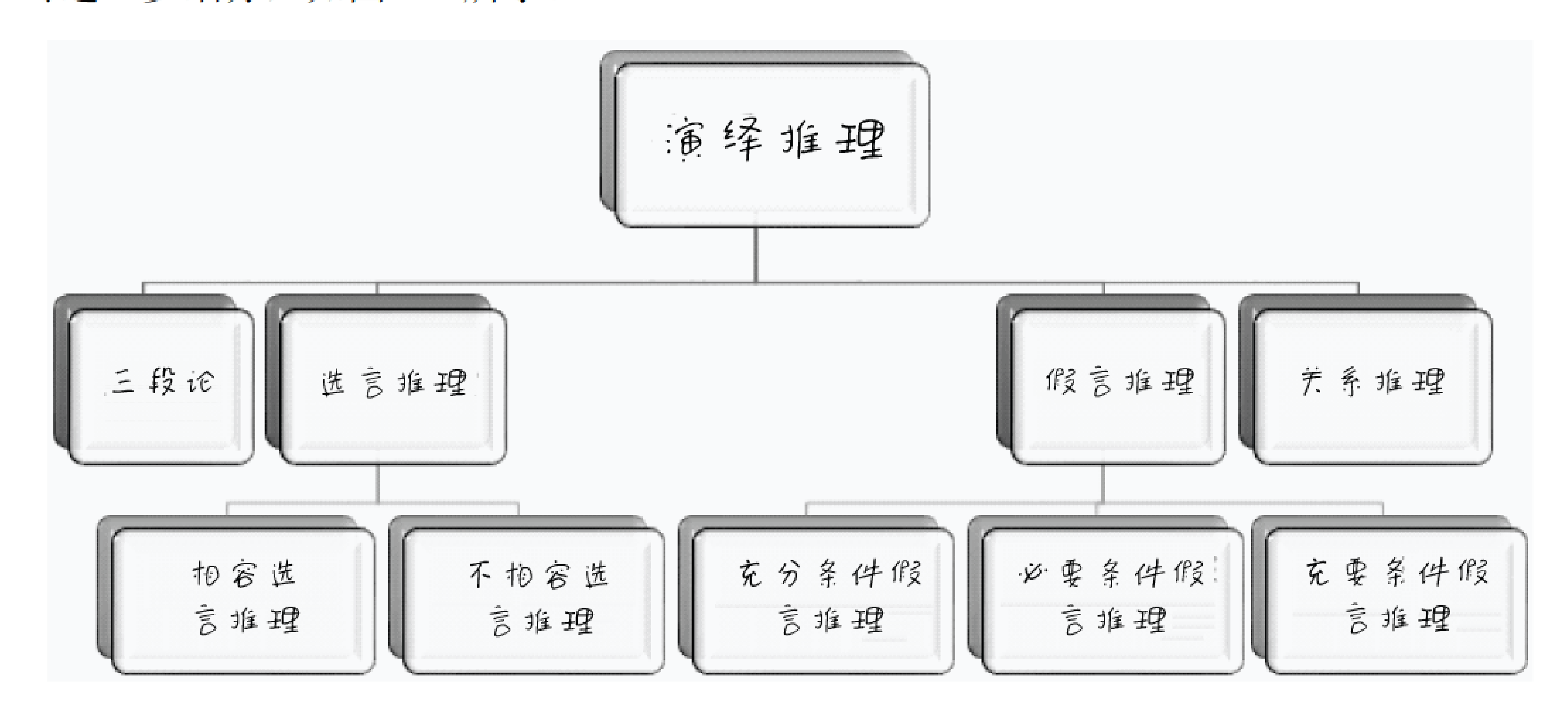


图 9-1

9.1.2　三段论

三段论推理是演绎推理中的一种简单判断推理。它是由两个含有一个共同项的性质判断作为前提，得出一个新的性质判断为结论的演绎推理。

从思维过程来看，任何三段论都必须具有大前提、小前提和结论，缺少任何一部分就无法构成三段论推理，各部分的含义如下。

- ❑ 大前提：已知的一般原理。
- ❑ 小前提：所研究的特殊情况。
- ❑ 结论：根据一般原理，对特殊情况作出判断。

我们首先来看一个例子：

参加政协会议的都是政协委员。
王五参加了政协会议。
王五是政协委员。

在上面的例子中，大前提是“参加政协会议的都是政协委员”，小前提是“王五参加了政协会议”，结论是“王五是政协委员”。

一个正确的三段论有且仅有 3 个词项。

- ❑ 大项（用 P 表示）：出现在大前提中，又在结论中做谓项的词项。

❑ 中项（用 M 表示）：联系大小前提的词项。

❑ 小项（用 S 表示）：出现在小前提中，又在结论中做主项的词项。

在上面的例子中，大项 P 是“政协委员”，中项 M 是“政协会议”，小项 S 是“王五”。

可以看出，三段论的演绎过程如图 9-2 所示。

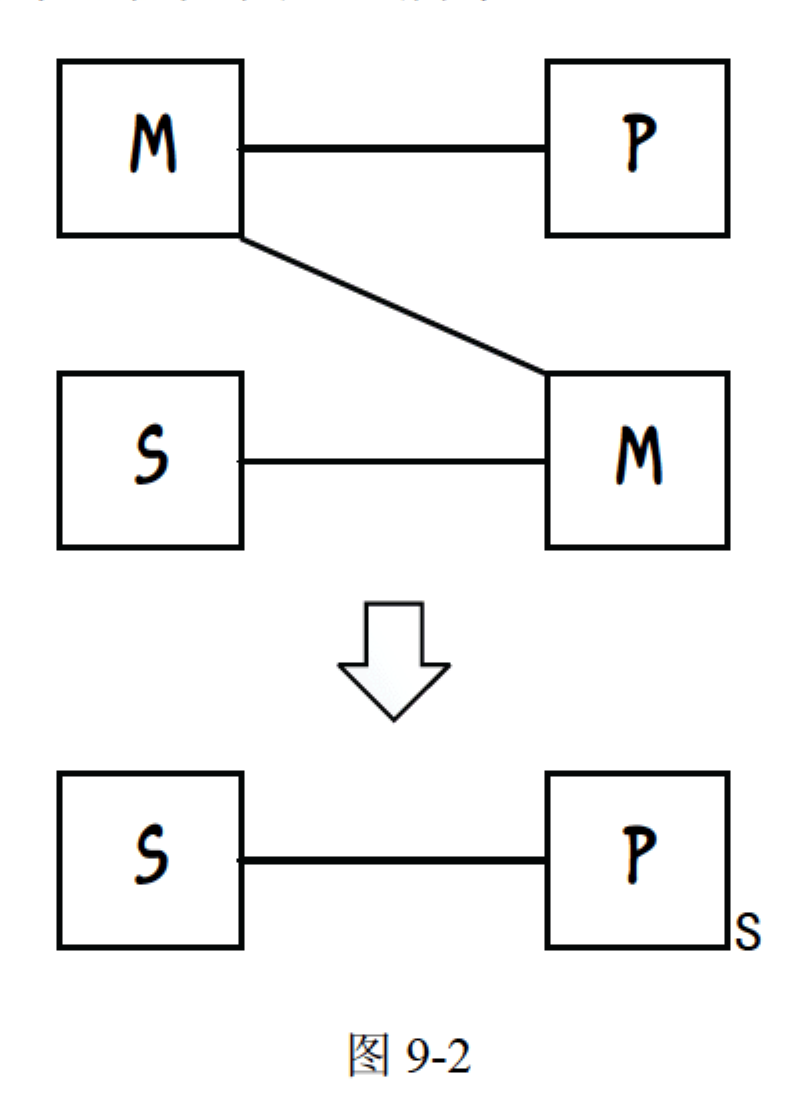


图 9-2

从图 9-2 和前面的文字描述可以看出，三段论推理是根据两个前提所表明的中项 M 与大项 P 和小项 S 之间的关系，通过中项 M 的媒介作用，从而推导出确定小项 S 与大项 P 之间关系的结论。还可看出，在三段论中每个项（P、M、S）都出现了两次。

需要注意的是，必须使三段论中的三个项（P、M、S）在其分别重复出现的两次中，所指的是同一个对象，具有同一个外延。否则就会犯四概念的错误。所谓四概念的错误就是指，在一个三段论中出现了 4 个不同的概念。四概念的错误往往是由于作为中项的概念未保持同一而引起的。

例如：

中国的大学分布于全国各地。

清华大学是中国的大学。

清华大学分布于全国各地。

首先，这个三段论的结论显然是错误的！可是，这个三段论的大小前提都是真的，为什么会由两个真的前提推出一个假的结论来了呢？

我们先来分解一下大、中、小项。

❑ 大项 P：全国各地；

❑ 中项 M：中国的大学；

❑ 小项 S：清华大学。

在大前提（中国的大学分布于全国各地）中，中项 M“中国的大学”表示的是中国的大学总体，是一个集合概念。

而在小前提中，中项 M“中国的大学”表示的是“（清华大学是）中国的大学（中的一所）”，后面省略了部分内容，在这里表示的不是集合概念，而是一个一般的普遍概念。

因此，看起来文字是一样的两次重复出现的内容“中国的大学”，表示的实质不是同一个概念。也就是说，中项 M（“中国的大学”）未保持同一，就出现了 4 个概念，不再只有大项、中项、小项这 3 个概念了。这样，以“中国的大学”这个具有不同概念的内容作为中项，也就无法将大项和小项必然地联系起来，从而推出正确的结论。

9.1.3　选言推理

选言推理是指传统逻辑里一类有两个前提的演绎推理，其中一个前提是选言命题，另一个是该选言命题的支命题的负命题。

什么是选言命题呢？看一个例子：

购房时可以按揭付款，也可以全额付款。

上面的命题中包含两个（或多个）选择，就称为选言命题。

选言命题有相容与不相容之分，相容选言命题是指选言中的两个选言支是相容的，即都可以选择。而不相容选言命题是指两个选言支是不相容的，只能选择其中一支。如上面例子中“购房付款”的两种方式就是不相容的。

相应地，选言推理分为相容选言推理和不相容选言推理两种。

1．相容选言推理

相容选言推理就是以相容选言命题为前提，根据相容选言命题的逻辑性质进行的推理。

相容选言推理的基本原则是：大前提是一个相容的选言判断，小前提否定了其中一个（或一部分）选言支，结论就要肯定剩下的一个选言支。但是，若肯定一部分选言支，并不能否定另一部分选言支。也就是说，有以下两条规则：

❑ 否定一部分选言支，就要肯定另一部分选言支。

❑ 肯定一部分选言支，不能否定另一部分选言支。

根据规则，相容选言推理只有一个正确的形式，即否定肯定式。就是在大前提中确定相容的选言支，在小前题中否定一部分选言支，则可得出肯定另一部分选言支的结论，

具体流程如图 9-3 所示。

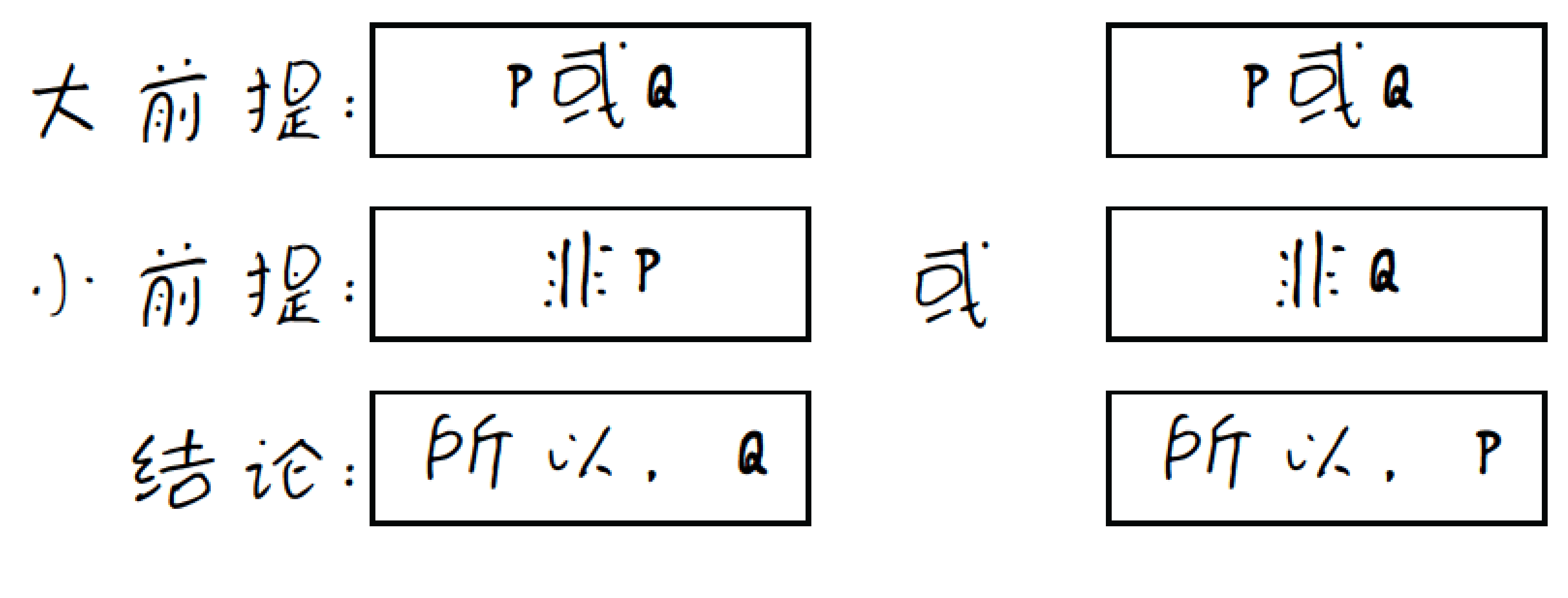


图 9-3

例如：有如图 9-4 所示的选言推理。

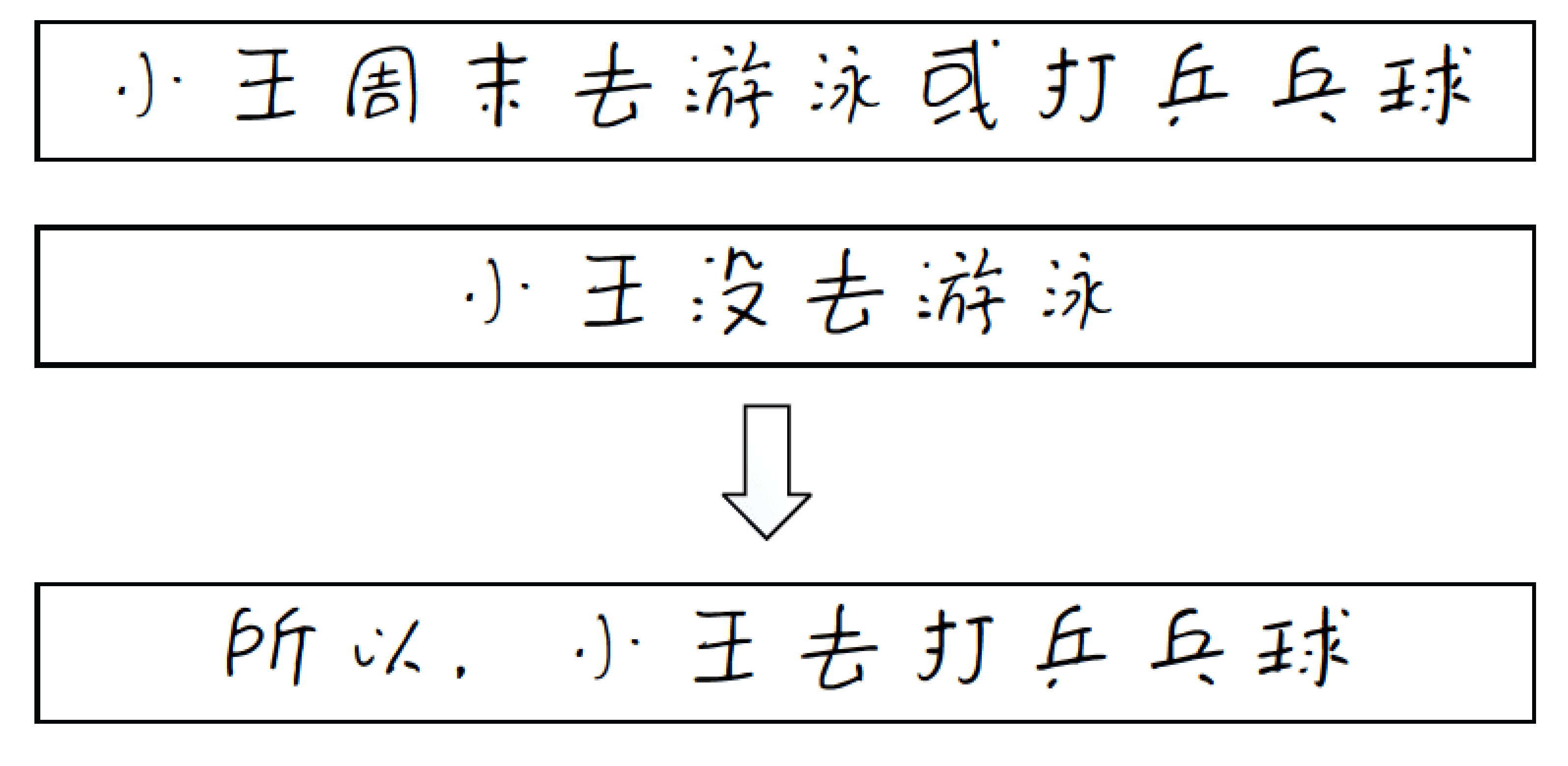


图 9-4

在上面的例子中，“游泳”和“打乒乓球”是相容的，否定一部分选言支“游泳”，可以肯定另一部分选言支“打乒乓球”。因此，上面的选言推理是正确的。

但是，如果肯定一部分选言支，并不能否定另一部分选言支。例如，如图 9-5 所示的推理就是错误的。在这个推理中，肯定了一部分选言支（“游泳”），并不能否定另一部分选言支（“打乒乓球”）。因为相容选言命题的选言支“游泳”和“打乒乓球”可以同时为真。也就是说，既可去“游泳”，又去“打乒乓球”。因此，肯定“游泳”，不能否定“打乒乓球”。

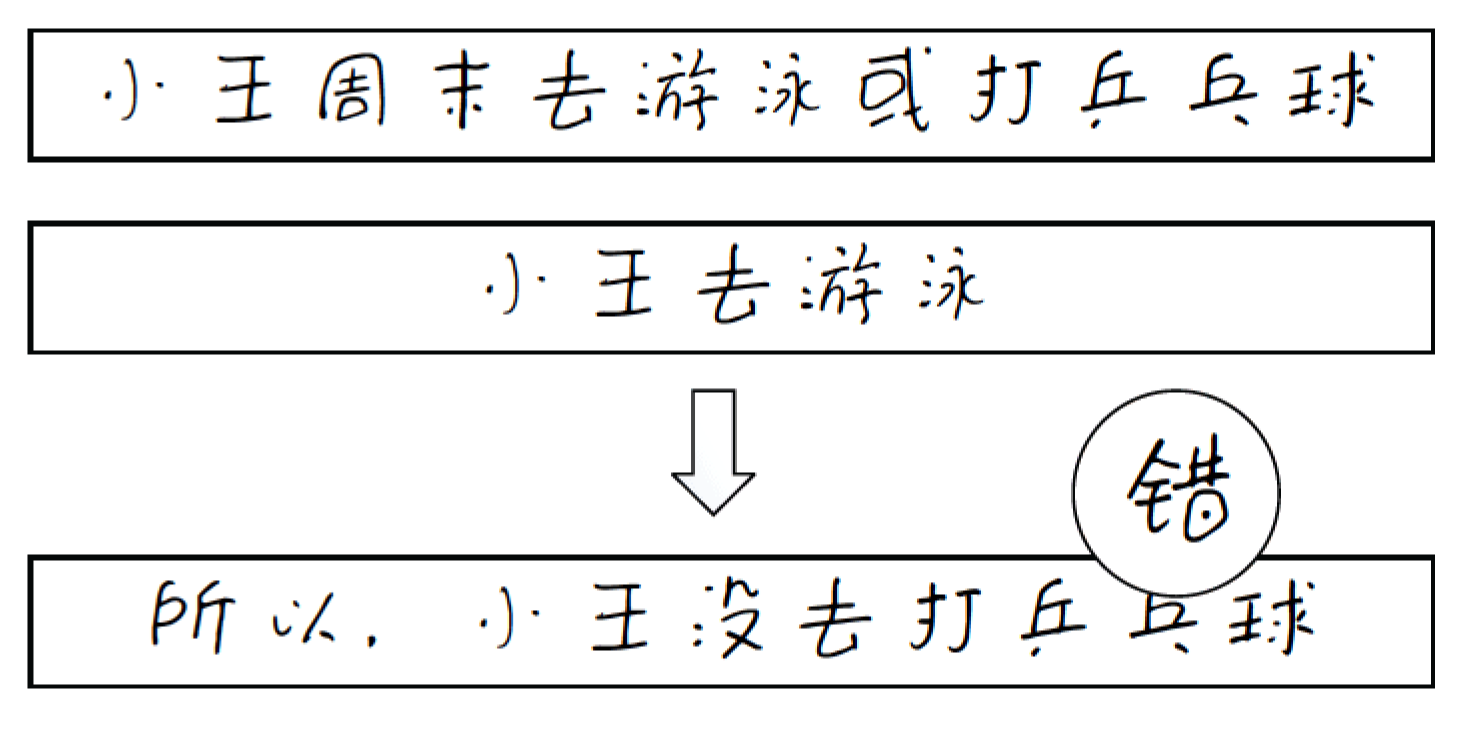


图 9-5

2. 不相容选言推理

不相容选言推理就是以不相容选言命题为前提，根据不相容选言命题的逻辑性质进行的推理。

与相容选言推理不同，由于不相容选言中的选言支是不相容的，因此，否定一部分选言支，就会肯定另一部分选言支（这点与相容选言推理相同）；并且，肯定一部分选言支，就要否定另一部分选言支（这点与相容选言推理不同）。也就是说，不相容选言推理有以下两条规则：

- ❑ 否定一部分选言支，就要肯定另一部分选言支。
- ❑ 肯定一部分选言支，就要否定另一部分选言支。

根据规则，不相容选言推理有两个正确的形式，一种是否定肯定式（与相容选言推理相同），参考图 9-3。

另一种是肯定否定式（与相容选言推理不相同），就是在大前提中确定不相容的选言支，在小前提中肯定一部分选言支，则可得出否定另一部分选言支的结论，具体流程如图 9-6 所示。

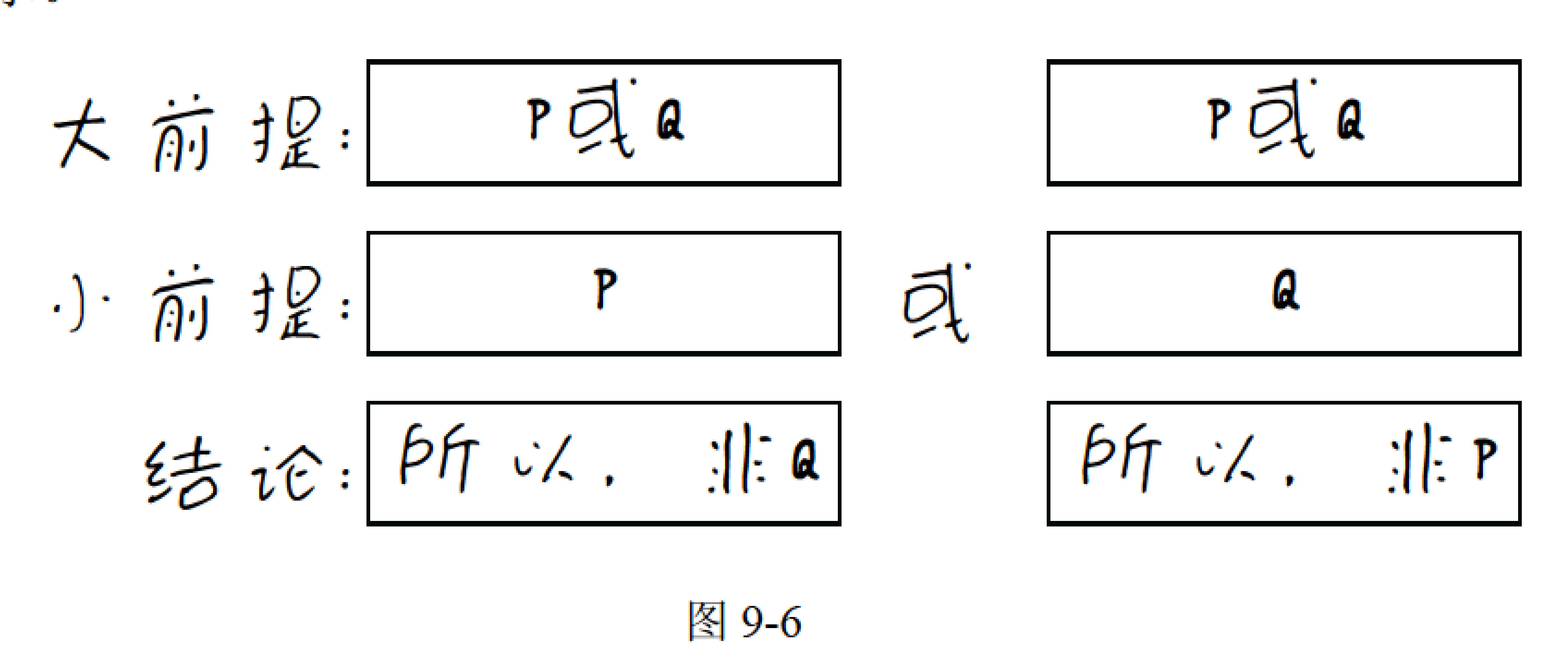


图 9-6

例如，有如图 9-7 所示的选言推理。

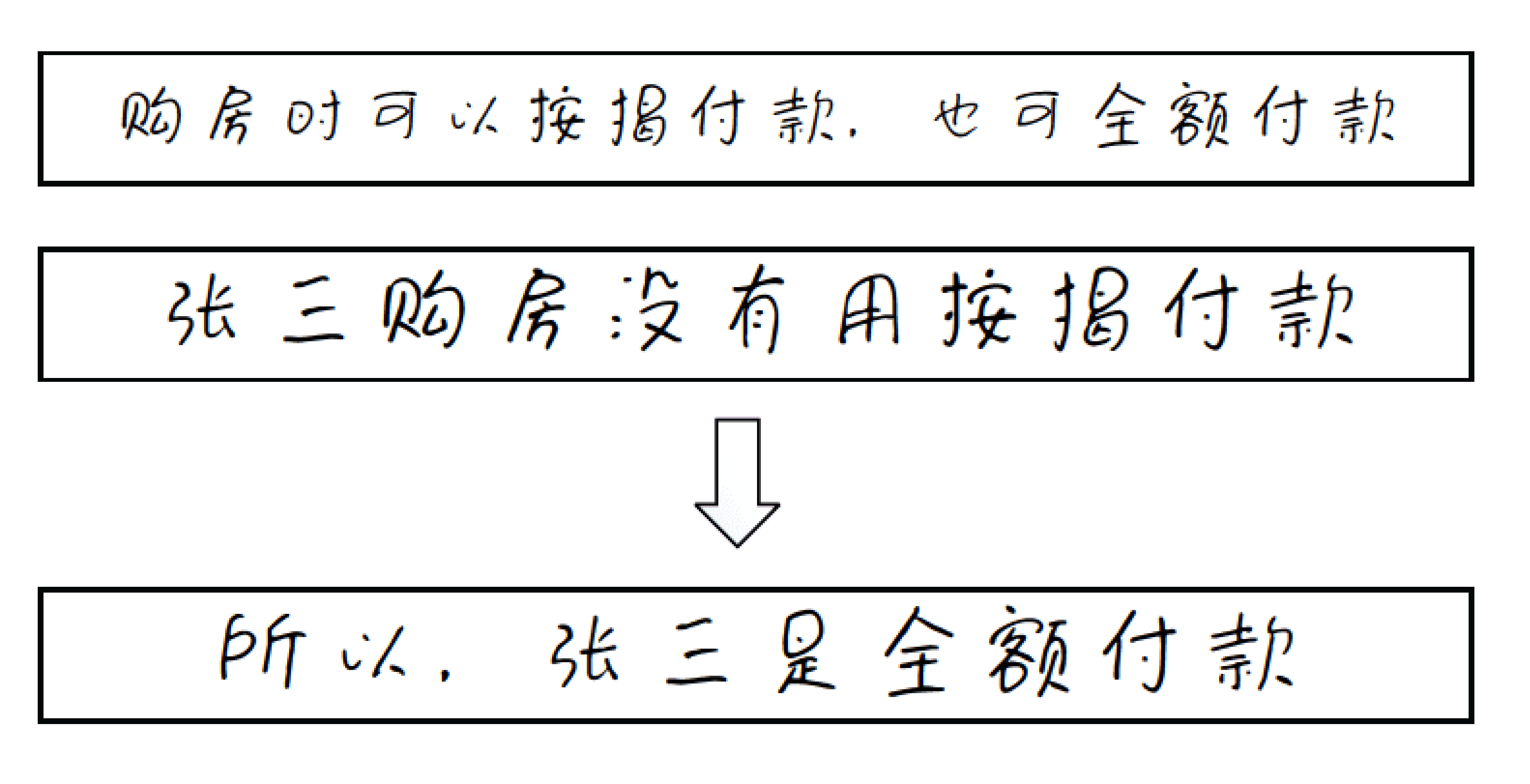


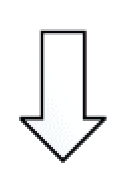
图 9-7

图 9-7 所示选言推理是采用“否定肯定式”，即先否定“按揭付款”，可推理出“全额付款”为真的结论。而图 9-8 则是先肯定“按揭付款”，从而推理出“全额付款”为假

的结论。

购房时可以按揭付款，也可全额付款

张三购房是按揭付款



所以，张三未全额付款

图 9-8

9.1.4 假言推理

假言推理是根据假言命题的逻辑性质进行的推理。有关假言命题的相关内容可参见本书 8.2.5 节中的内容。

根据 3 种不同的假言命题，假言推理可分为充分条件假言推理、必要条件假言推理和充分必要条件假言推理等 3 种。

1. 充分条件假言推理

充分条件假言推理是根据充分条件假言命题的逻辑性质进行的推理。充分条件假言推理有两条规则：

- ❑ 肯定前件，就要肯定后件；否定前件，不能否定后件。
- ❑ 否定后件，就要否定前件；肯定后件，不能肯定前件。

在充分条件假言推理中，由前件可以推出后件，但并不能由后件推出前件。

根据规则，充分条件假言推理有两个正确的形式。根据第 1 条规则，可得到“肯定前件式”，如图 9-9 所示。

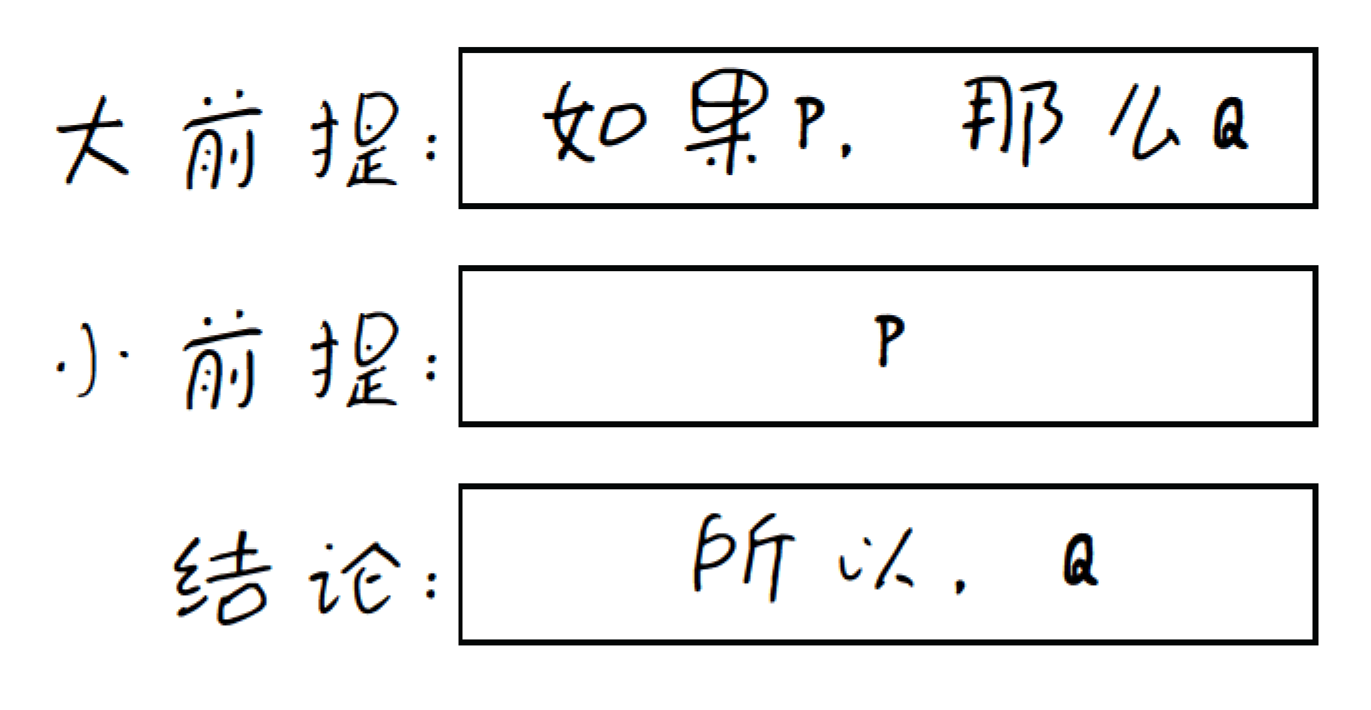


图 9-9

充分条件假言推理的另一种形式是“否定后件式”，如图 9-10 所示。

大前提：如果P，那么Q

小前提：非Q

结论：所以，非P

图 9-10

下面我们来看两个例子，首先看一下“肯定前件式”的例子：

如果两个三角形全等，则它们面积相等；这两个三角形全等，所以，它们面积相等。

在这个例子中，前件是“两个三角形全等”，后件是“两个三角形面积相等”，肯定了前件（两个三角形全等），因此，后件也必定为真（面积相等）。但是，如果否定前件，并不能否定后件，即“两个三角形不全等，并一定其面积就不相等”。

用同一个例子来看“否定后件式”：

如果两个三角形全等，则它们面积相等；这两个三角形面积不等，所以，它们不是全等三角形。

否定了后件（两个三角形面积不等），因此，也将否定前件（不是全等三角形）。但是，如果肯定后件，并不能肯定前件，即“两个三角形面积相等，并一定就全等”。

2．必要条件假言推理

必要条件假言推理是根据必要条件假言命题的逻辑性质进行的推理。必要条件假言推理有两条规则：

- ❑ 否定前件，就要否定后件；肯定前件，不能肯定后件。
- ❑ 肯定后件，就要肯定前件；否定后件，不能否定前件。

根据规则，必要条件假言推理有两个正确的形式。根据第一条规则，可得到“否定前件式”，如图 9-11 所示。

大前提：只有P，才Q
小前提：非P
结论：所以，非Q

图 9-11

另一种形式是“肯定后件式”，如图 9-12 所示。

大前提：只有P，才Q
小前提：Q
结论：所以，P

图 9-12

例如：下面是“否定前件式”：

只有年满十八周岁，才有选举权，小李未满十八岁，所以，小李没有选举权。

这个例子中，前件是“年满十八周岁”，后件是“选举权”，否定了前件，所以后件也为假。但是，肯定前件并不能肯定后件，即“小李满了十八周岁，并不一定就有选举权”，因此“满十八周岁”只是“选举权”的一个必要条件，其他条件也可能导致没有“选举权”。

下面的例子是“肯定后件式”：

只有年满十八周岁，才有选举权，小张有选举权，所以，小张年满十八岁了。

肯定了后件（有选举权），则前件必定为真（年满十八岁），也就是说“只要有选举权，肯定已满十八周岁”。但是，否定后件不一定能否定前件，如“小李没有选举权”，并不一定表示“小李未满十八周岁”，因为还有其他原因导致小李没有选举权，而“年满十八周岁”只是一个必要条件。

3．充分必要条件假言推理

充分必要条件假言推理是根据充分必要条件假言命题的逻辑性质进行的推理。充分必要条件假言推理有两条规则：

- ❑ 肯定前件，就要肯定后件；肯定后件，就要肯定前件。
- ❑ 否定前件，就要否定后件；否定后件，就要否定前件。

根据规则，充分必要条件假言推理有 4 种正确的形式。根据第一条规则，有“肯定前件式”和“肯定后件式”两种形式，如图 9-13 所示。

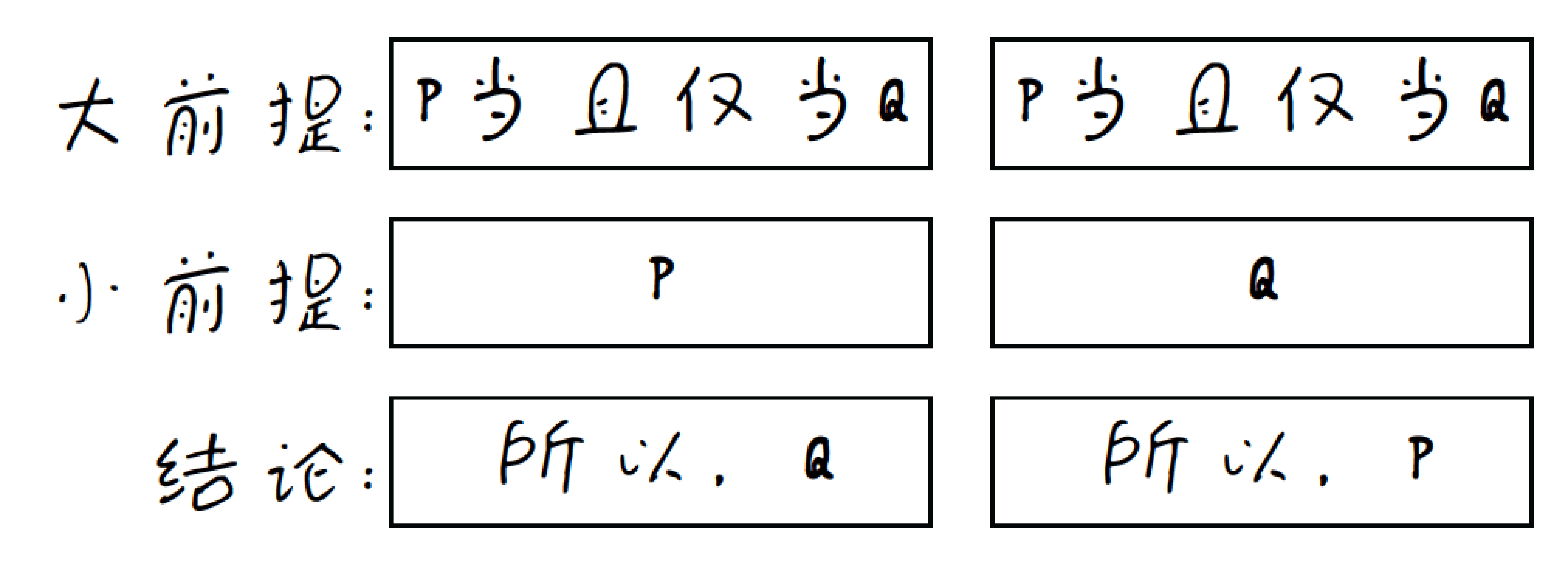


图 9-13

根据第二条规则有“否定前件式”和“否定后件式”两种形式，如图 9-14 所示。

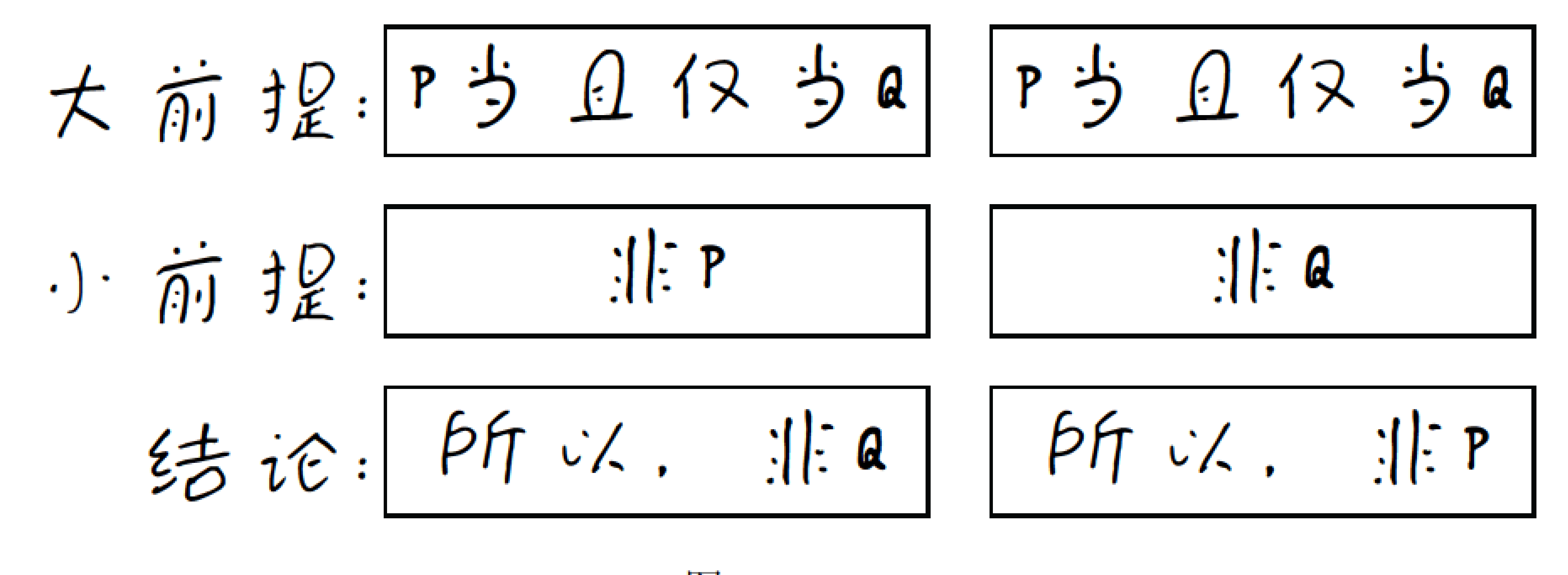


图 9-14

例如：以下例子是“肯定前件式”。

一个数当且仅当能被2整除，这个数为偶数；10能被2整除，所以，10是偶数。

在上面例子中，前件是（能被 2 整除），后件为（是偶数），当肯定前件（能被 2 整除），则后件也为真（是偶数）。

而“肯定后件式”例子如下：

一个数当且仅当能被2整除，这个数为偶数：10是偶数，所以，10能被2整除。

这个例子肯定后件（是偶数），则前件也为真（能被 2 整除）。

类似地，如下是“否定前件式”的例子：

一个数当且仅当能被2整除，这个数为偶数：9不能被2整除，所以，9不是偶数。

这个例子否定前件（不能被 2 整除），所以，后件也为假（不是偶数）。

再看“否定后件式”的例子：

一个数当且仅当能被2整除，这个数为偶数：9不是偶数，所以，9不能被2整除。

这个例子否定后件（不是偶数），所以，前件也为假（不能被 2 整除）。

9.1.5　关系推理

在演绎推理中，关系推理是比较简单的一种类型，关系推理没有太多的分类。

所谓关系推理，是指前提至少有一个是关系判断，并按其关系的逻辑性质而进行推演的演绎推理。

例如：有“A 大于 B”的一个关系判断，则可以有以下关系推理：

前提：A大于B

结论：B不大于A

这个例子是一种反对称性关系推理。就是根据一个已知的关系判断，反向推理出另一个关系判断。

另外，对于在前提中定义的多个关系判断，还可能会有传递。例如，有“A 大于 B，B 大于 C”这样两个关系判断作为前提，则可推理出：

前提：A大于B，B大于C

结论：A大于C

9.1.6　演绎推理综合实例

在日常生活中我们经常要用到演绎推理的一种或几种形式。例如，在公务员考试中就经常会有这方面的试题，下面我们来看一道公务员考试中的演绎推理试题，具体题目如下：

某仓库失窃，4 个保管员因涉嫌而被传讯。4 人的供述如下：

甲：我们 4 人都没作案；

乙：我们中有人作案；

丙：乙和丁至少有一人没作案；

丁：我没作案。

如果 4 人中有两人说的是真话，有两人说的是假话，则以下哪项断定成立？

A．说真话的是甲和丁　　　　B．说真话的是乙和丙

C．说真话的是甲和丙　　　　D．说真话的是乙和丁

这道题的前提是：4 个人中有两人说真话，两人说假话。

下面就来分析推理哪两人说的是真话，哪两人说的是假话，具体过程如下。

首先看甲，他说的是“我们 4 人都没有作案”，他这话的含义是“作案者不在我们 4 个人中，另有其人”，他的话有可能真，也可能假。

接着看乙，他说的是“我们中有人作案”，他这话的含义是“作案者在我们 4 人中”，他的话有可能真，也有可能假。

与甲的话对比来看，两人的话是互相矛盾的，如果一个说的是真话，另一个肯定就是假话。也就是说，甲、乙两人中有一人说真话，有一人说假话，即甲、乙之间构成了不相容选言推理。

如果甲，则非乙

如果乙，则非甲

根据前提（有两人说真话，两人说假话），既然甲、乙中有一人说假话了，那么，另一个说假话的人就是丙和丁中的一位了。这构成一个假言推理：如果甲、乙有一人说假话，那么，丙、丁也有一人说假话。

大前提：两人说假话

小前提：甲乙有一人说假话

结论：丙丁有一人说假话

根据以上结论，可以得到丙、丁之间也是一种不相容选言推理，即：

如果丙，则非丁

如果丁，则非丙

再看丙，他说的是“乙和丁至少有一人没作案”，他这话的含义是“乙和丁至少有一人没作案，也可能两人都没作案”。

而丁说的是“我没作案。”

根据前面的推理，丙、丁之间有一人说假话，构成不相容选言推理。如果丁说的是真话，那么可以推出丙也说的是真话（因为丙说“乙和丁至少有一人没作案”），显然不符合前面推出的不相容选言推理的结论。

因此，可以推理出丁说的是假话，则丙说的就是真话。

接下来，由于丁说的是假话，丁说“我没有作案”，就是假的了，取否定值，则表示“丁作案了”，那么，很明显就可看出甲、乙两人中甲说的是假话（甲说“我们四人都没有作案”，而现在已确定丁是作案者），乙说的是真话了（乙说“我们中有人作案”）。

最后得出结论经，四人中，甲、丁说的是假话，乙、丙说的是真话，即选择答案B。

9.2 归纳推理

除了演绎推理外，在逻辑推理中常用的还有一种称为“归纳推理”的方法。下面我们来研究归纳推理，以及对比其与演绎推理的关系。

9.2.1 什么是归纳推理

我国著名数学家华罗庚写的《数学归纳法》一书中，举过这样一个例子：

从一个袋子里摸出来的第一个是红玻璃球，第二个是红玻璃球，甚至第三个、第四个、第五个都是红玻璃球的时候，我们立刻会出现一种猜想："是不是这个袋子里的东西全部都是红玻璃球？"但是，当我们有一次摸出一个白玻璃球的时候，这个猜想失败了。这时，我们会出现另一种猜想："是不是袋里的东西全都是玻璃球？"但是，当有一次摸出来的是一个木球的时候，这个猜想又失败了。那时，我们又会出现第三个猜想："是不是袋里的东西都是球？"这个猜想对不对，还必须继续加以检验，要把袋里的东西全部摸出来，才能见分晓。

华罗庚举的这个例子，是对简单枚举归纳推理结论性质的一个通俗说明。

人们应用简单枚举归纳推理，当然可以从为数不多的事例中推导出普遍的规律性来，然而这还是一个"猜想"。这种猜想对不对，还必须进一步加以验证。

从一个袋子里摸球，连续摸了 5 次，摸的都是红玻璃球，这时候，我们可以通过简单枚举归纳推理得出结论："这个袋子里装的都是红玻璃球。"但是，得出这个结论时，必须清醒地认识到这个结论是不可靠的。正如这个例子所表明的，第 6 次摸出的却是白玻璃球了，这就把前面的结论推翻了。因此，当摸了 6 个球后，只能得出"这个袋子里装的都是玻璃球"的结论了；摸第 7 个球时，又只能得出"这个袋子里装的都是球"的结论。当然，这个结论也未必正确。

现在，我们可以给归纳推理下个定义了。所谓归纳推理，就是从个别性知识推出一般性结论的推理。例如，在上面的摸球例子中，就是从有限的摸球次数中推出相应的结论。

传统上，根据前提所考察对象范围的不同，把归纳推理分为完全归纳推理和不完全归纳推理，如图 9-15 所示。完全归纳推理考察了某类事物的全部对象，不完全归纳推理则仅仅考察了某类事物的部分对象。例如，在前面摸球的例子中，经过多次摸球得出结论是不完全归纳推理，如果将袋中的球全部摸出后得出的结论就是完全归纳推理。

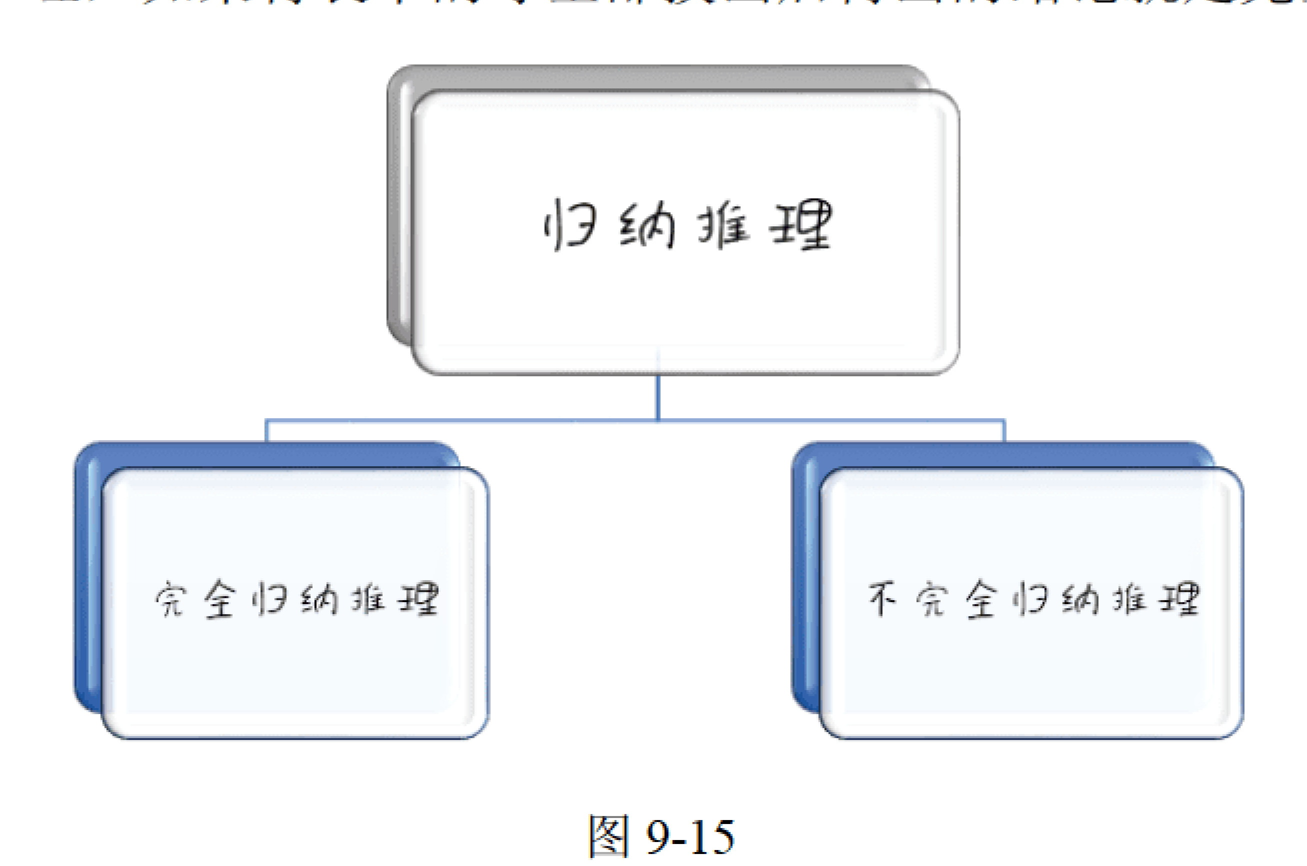


图 9-15

归纳推理的前提是其结论的必要条件。另外，归纳推理的前提是真实的，但结论却未必为真，而可能为假。例如，有名的"守株待兔"故事，就是根据某天有一只兔子撞到树上死了，推出每天都会有兔子撞到树上死掉，显然这个结论是假的。

9.2.2 完全归纳推理

完全归纳推理是根据某类事物中每 1 对象都具有（或不具有）某种属性，从而推出该类事物全部对象都具有（或都不具有）某种属性的结论。

例如：以下就是一个完全归纳推理的例子。

太平洋已经被污染；

大西洋已经被污染；

印度洋已经被污染；

北冰洋已经被污染；

太平洋、大西洋、印度洋和北冰洋是地球上的全部大洋。

所以，地球上所有大洋都已经被污染。

可以看到，这种完全归纳推理的逻辑形式如下：

S_1 是 P；

S_2 是 P；

…… ……

S_n 是 P；

S_1、S_2、…、S_n 是 S 类的全部对象。

所以，所有 S 都是 P。

完全归纳推理的特点是：在前提中考察了某一类事物的全部对象，结论没有超出前提所断定的知识范围，因此，其前提和结论之间的联系是必然的。

运用完全归纳推理要获得正确的结论，必须满足两条要求：

- 在前提中考察了某一类事物的全部对象。
- 前提中对该类事物每个对象所做出的断定都是真的。

完全归纳推理在日常生活中经常用到。如“某中学的实验班高考成绩都上了二本线”、“今天车间生产的产品全部合格”、“实验班的任课老师都是高级教师”等结论，都是通过完全归纳推理获得的。概括地说，完全归纳推理的作用主要有以下两方面：

- 认识作用。虽然完全归纳推理的前提所断定的知识范围和结论所断定的知识范围相同，但它仍然可以提供新知识。这是因为，它的前提是个别性知识的判断，而结论则是一般性知识的判断，也就是说，完全归纳推理能使认识从个别上升

到一般。

- 论证作用。由于完全归纳推理是一种前提蕴涵结论的必然性推理，因而人们常常用它来证明论点。

由于完全归纳推理的结论必须在考察了某类事物的全部对象后才能做出，因此这种推理方法的使用会受到一定的限制。例如，通常以下两种情况就无法（或不适合）使用完全归纳推理：

- 当对某类事物中包含的个体对象的确切数目还不甚明了，或遇到该类事物中包含的个体对象的数目太大乃至无限大，没办法一一考察时，这时，就不可能使用完全归纳推理。
- 当某类事物中包含的个体对象虽有限，也能考察穷尽，但不宜考察或不必考察时，也不适宜使用完全归纳推理。例如，某鞭炮厂要考察本批产品是否全部能正常燃放，不可能将所有鞭炮都燃放了，再得出结论。

9.2.3　不完全归纳推理

前面提到，当所要考察的事物数量极多，甚至是无限的时候（或者因为某些特殊情况），不能使用完全归纳推理，这时就需要使用不完全归纳推理。那么，什么是不完全归纳推理？

从名称可看出，不完全归纳推理是根据某一类事物中的一部分对象都具有某种属性，从而推出该类事物都具有该种属性的结论。

在进行不完全归纳推理时，根据选择某一类事件中一部分对象的不同方法，又可将不完全归纳推理分为简单枚举归纳推理、科学归纳推理、概率归纳推理和统计归纳推理等 4 种方式。

1．简单枚举归纳推理

在一类事物中，根据已观察到的部分对象都具有某种属性，并且没有遇到任何反例，从而推出该类事物都具有该种属性的结论，这就是简单枚举归纳推理。

例如，哥德巴赫猜想就是用了简单枚举归纳推理提出来的。200 多年前，德国数学家哥德巴赫发现一个现象，一些奇数都分别等于 3 个素数之和，例如：

17=3+3+11

41=11+13+17

77=7+17+53

461=5+7+449

哥德巴赫并没有把所有奇数都列举出来（也不可能将所有奇数列完），只是从少数例子出发就提出了一个猜想：所有大于 5 的奇数都可以分解为 3 个素数之和。这就是典

型的“简单枚举归纳推理”。

可以看出，简单枚举归纳推理的逻辑形式如下：

S_1是P；

S_2是P；

…　…

S_n是P；

S_1、S_2、…、S_n是S类的部分对象，且不存在S_i不是P的情况。

所以，所有S都是P。

可以看出，简单枚举归纳推理的结论是或然的（不是必然的）。如本节开头介绍的华罗庚所说的摸球例子，随着枚举对象数量的增多，有可能出现反例。

因此，要提高简单枚举归纳推理的可靠性，必须注意以下两点：

- 枚举的数量要足够多，考察的范围要足够广。
- 考察有无反例。

在进行枚举归纳推理时，如果不注意以上两点，就会犯“以偏盖全”的逻辑错误，出现类似“守株待兔”式的结论。

2．科学归纳推理

科学归纳推理是根据某一类事物中部分对象与某种属性间因果联系的分析，推出该类事物具有该种属性的推理。

例如，有以下实验及推理过程：

铁受热后体积膨胀；

铜受热后体积膨胀；

铝受热后体积膨胀；

由于金属受热后分子凝聚力减弱，分子运动加速，分子间距离增加，从而导致体积膨胀，而铁、铜、铝都是金属；

所以，所有金属受热体积都膨胀。

可以看出，前面部分与枚举归纳类似，都是枚举某一类事物的同一个属性，只是在有限的枚举之后，有一段关于属性与对某一类对象之间有因果联系（而这个因果联系是与科学分析相关的）的描述，最后得出结论。

因此，科学归纳推理的逻辑形式如下：

S_1 是 P；

S_2 是 P；

…　…

S_n 是 P；

S_1、S_2、…、S_n 是 S 类的部分对象，且与 P 有因果联系。

所以，所有 S 都是 P。

需要注意的是，在科学归纳推理中，枚举出某一类事物中的对象与属性之间的因果联系，必须是满足已有科学知识（如上例中“分子受热”变化是一种客观存在的科学知识）。

3. 概率推理

在本书第 6 章专门讨论了概率，并且知道概率是一种数学统计方法。其实，也可以将这种数学统计规律运用到逻辑推理中，就形成了概率推理这种方法。

根据第 6 章介绍的知识我们知道，某种随机事件的概率愈大，表明该事件发生的可能性程度就愈大；反之，其概率愈小，表明该事件发生的可能性程度也就愈小。因此，某一随机事件的概率大小，标志着该事件发生的可能性的大小。运用概率这种逻辑方法（它更是一种数学方法）进行逻辑推理时，首先需要对大量的基本事件进行广泛的考查。考查范围愈广，对象愈多，从中获得的概率本身的正确性就愈大；反之，如果考查范围很窄，对象很少，那么从中获得的概率，未必就是该类事件的概率。因此还可以说，概率是从个别中归纳出一种关于一般的可能性规律。

下面看一个例子：

根据概率相关知识，我们知道抛掷一枚硬币时，正面朝上和反面朝上的概率几乎相等。则连续抛掷 100 次硬币，正面朝上的次数为多少次？

这个例子中，大前提是“硬币正反面概率相等”，那么“抛 100 次硬币”，则可推出结论是“正面朝上 50 次”。

大前提：硬币正反面概率相等

小前提：抛100次硬币

结论：正面朝上50次左右

需要注意的是，这个结论不是必然的，而只是一个可能值。

概率告诉我们的是大量选取中所发生的情况，并不能直接推导出下一次的准确结果。例如，将上例中的问题进行修改，具体描述如下：

根据概率相关知识，我们知道抛掷一枚硬币时，正面朝上和反面朝上的概率几乎相等。现在已经连续抛掷了 5 次硬币，每次都是正面朝上，那么，接着抛第 6 次时一定会反面朝上吗？

根据概率知识，我们知道硬币正面朝上的概率为 50%，但并不意味着每两次抛掷硬币都会得到一正一反的结果。在这个例子中，前 5 次虽然都得到正面朝上的结果，但这个结果并不会影响后面硬币的正反面结果。也就是说，每一次抛掷硬币都是独立事件，与之前抛掷的结果并没有关系。

不过，随着抛掷硬币次数的增多，硬币正面朝上和反面朝上之比将会趋近于 1:1（即各点 50%）。

4．统计推理

在统计学中，某一被研究领域的全部对象称为总体；从总体中抽选出来加以考察的那一部分对象称为样本。由样本具有某种属性推导出总体也具有某种属性的推理称为统计推理。

例如，某大学对大四学生是否考研进行了抽样调查，并根据抽样数据进行统计，得出以下统计推理过程：

对大四学生考研的抽样调查

第1位，准备考研；

第2位，不考研；

第3位，准备考研；

第4位，不考研；

第5位，不考研；

……

一共调查了100位大四学生，其中25%的学生准备考研；

所以，该校大四学生有25%准备考研。

可以看出，在统计推理中，通过某一小部分样本的数据统计结果推导出整体数据的相关结论。根据上面的过程，可知道统计推理的逻辑形式如下：

S_1是P；

S_2是P；

S_3不是P；

……

S_n是P；

S_1、S_2、…、S_n是从S类抽取的样本，其中n%的对象具有性质P；

所以，S类所有对象中有n%都具有性质P。

在统计推理中，要使最终推导出的结论可靠，抽样必须要具有代表性，也就是说抽取的样本要具有代表性。根据不同的统计总体特性，可设计不同的抽样方式。例如，对于流水线上的产品进行抽样，可考虑在几个间隔均匀的时间段抽样，或间隔均匀的产品数量中进行抽样。

9.3　足球比赛的得分

学习了一些常用逻辑推理的方法，下面我们进行综合运用，推算一场比赛中各队之间的比分。

9.3.1　粗心的记分员

某大学数学系举办一次足球比赛，分年级组织了 4 队，分别是大一队、大二队、大三队、大四队。这 4 支足球队之间进行循环赛，每两队之间都比赛一场，即每支球队要与另外 3 队进行比赛，也就是说每支球队要踢 3 场比赛。

每场比赛后，比赛组织者都要将该场比赛中两队的比分填到如图 9-16 所示的比分表中，并将该场比赛后各队的输赢场数、进球数、失球数进行汇总填到如图 9-17 所示的表中。

比赛球队	比分
大四：大三	
大四：大二	
大四：大一	
大三：大二	
大三：大一	
大二：大一	

图 9-16

球队	已赛场数	获胜场数	失败场数	平局场数	进球数	失球数
大四						
大三						
大二						
大一						

图 9-17

可是，在进行了 5 场比赛之后，由于记分员小朱同学的粗心大意，将图 9-16 所示的比分表弄丢了，并且图 9-17 所示比赛成绩汇总表也被撕毁，最后只恢复出 3 行数据，如图 9-18 所示就是恢复的 3 行数据。

球队	已赛场数	获胜场数	失败场数	平局场数	进球数	失球数
大四	2	1	0	1	3	2
大三	3	2	0	1	2	0
大二	2	0	2	0	3	5

图 9-18

这可怎么办！眼看只剩一场比赛，整个循环赛就结束了，而前几周进行的这 5 场比赛也记不清各队的比分情况了。

怎么办？能不能用图 9-18 所示的残余数据推算出前 5 场比赛的比分？

9.3.2　从已有数据推算出比分

从图 9-18 所示的数据中可以看出，虽然这张表格只剩部分数据，不过好在只少了一行数据（大一队的数据不见了），还有 3 行数据，即 75%的数据保留下来了，应该可以通过这 3 行数据推导出被撕掉那一行的数据，同时，也可将各队的比分还原出来。

为了方便描述，为 4 支球队进行编号如下。

A：大四队；

B：大三队；

C：大二队；

D：大一队。

要还原出数据，首先应从较多的信息着手。根据前提条件（已知条件），4 支球队进行循环比赛，每支球队应该要进行 3 场比赛，而从图 9-18 所示残余数据中可看到，B 队共进行了 3 场比赛，因此，我们先从 B 队入手。具体步骤如下。

（1）先看 B 队（大三队）的成绩，共比赛了 3 场（即与其他 3 队都已比赛完成），比赛成绩是胜 2 场平 1 场，进 2 球。一场比赛要想获胜，至少应该进 1 球，若双方都不进球，就成 0:0 的平局了。因此，B 队胜了 2 场，一共只进了 2 球，则每场球都只能是以进 1 球获胜，推理过程如图 9-19 所示。

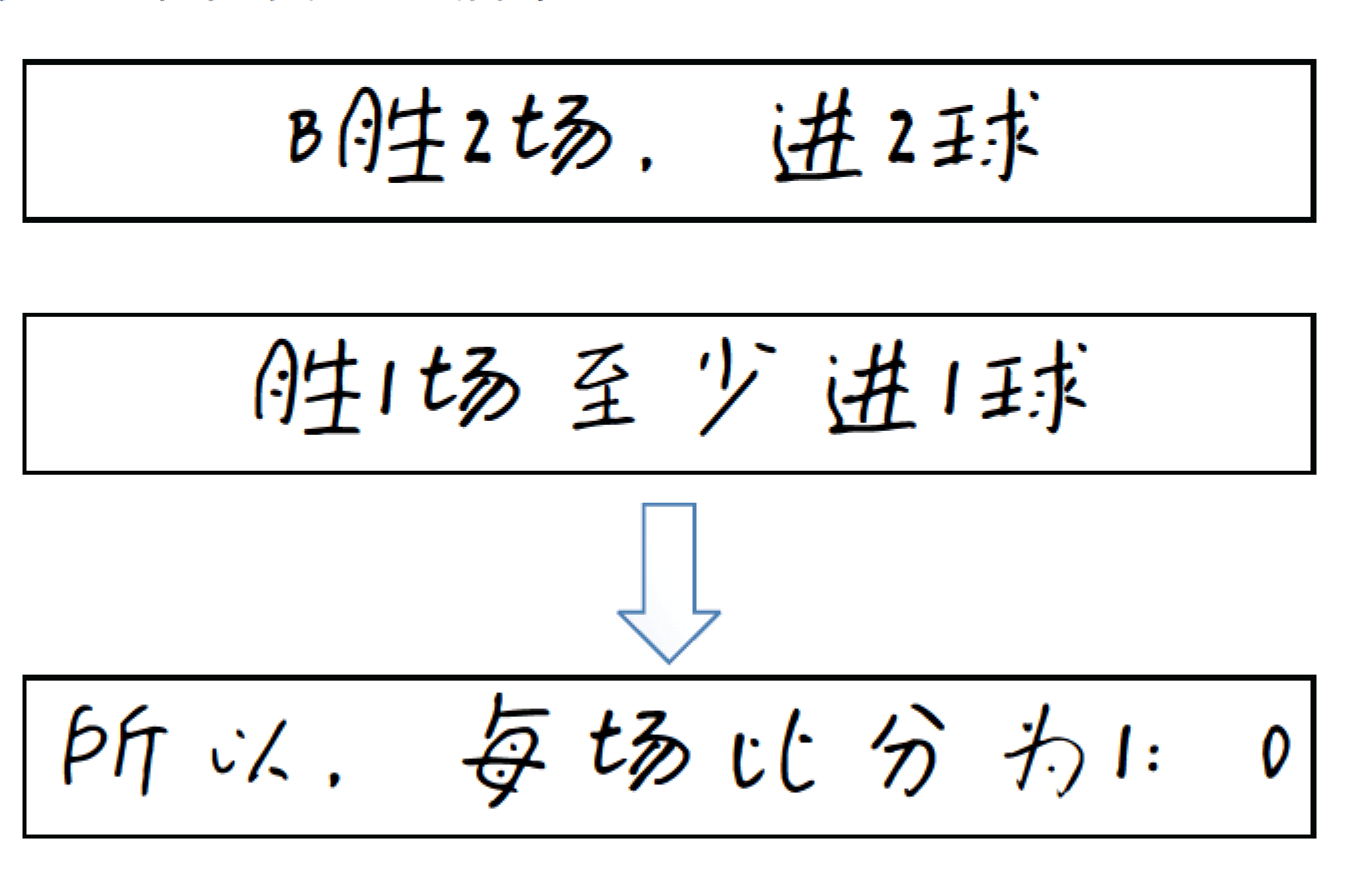


图 9-19

（2）而平局是指双方进球数量相等，B 队一共只进了 2 球，图 9-19 中可看到胜的 2 场已进了 2 球，因此，平 1 局时双方都未进球，即平 1 局的比分为 0:0，推理过程如图 9-20 所示。也就是说，B 队 3 场比赛的比分分别为 1:0、1:0、0:0。

B平1场

共进2球，失0球

胜2场进2球

↓

所以，平局比分为0:0

图 9-20

这时，还只是推理出 B 队 3 场比赛的比分，但还不知道具体和某一队的比分。

（3）再看 A 队（大四队）的成绩，共比赛了 2 场，胜 1 场平 1 场。由于 B 队已比赛完 3 场且一场未输（根据前面的推理），因此，A 队与 B 队应该是平局，且比分为 0:0，推理过程如图 9-21 所示。

（4）根据前面的推理，B 队 2 胜（比分分别为 1:0）1 平，现在知道与 A 队是平局，则与 C 队（大二队）、D 队（大一队）应该是以 1:0 获胜，推理过程如图 9-22 所示。

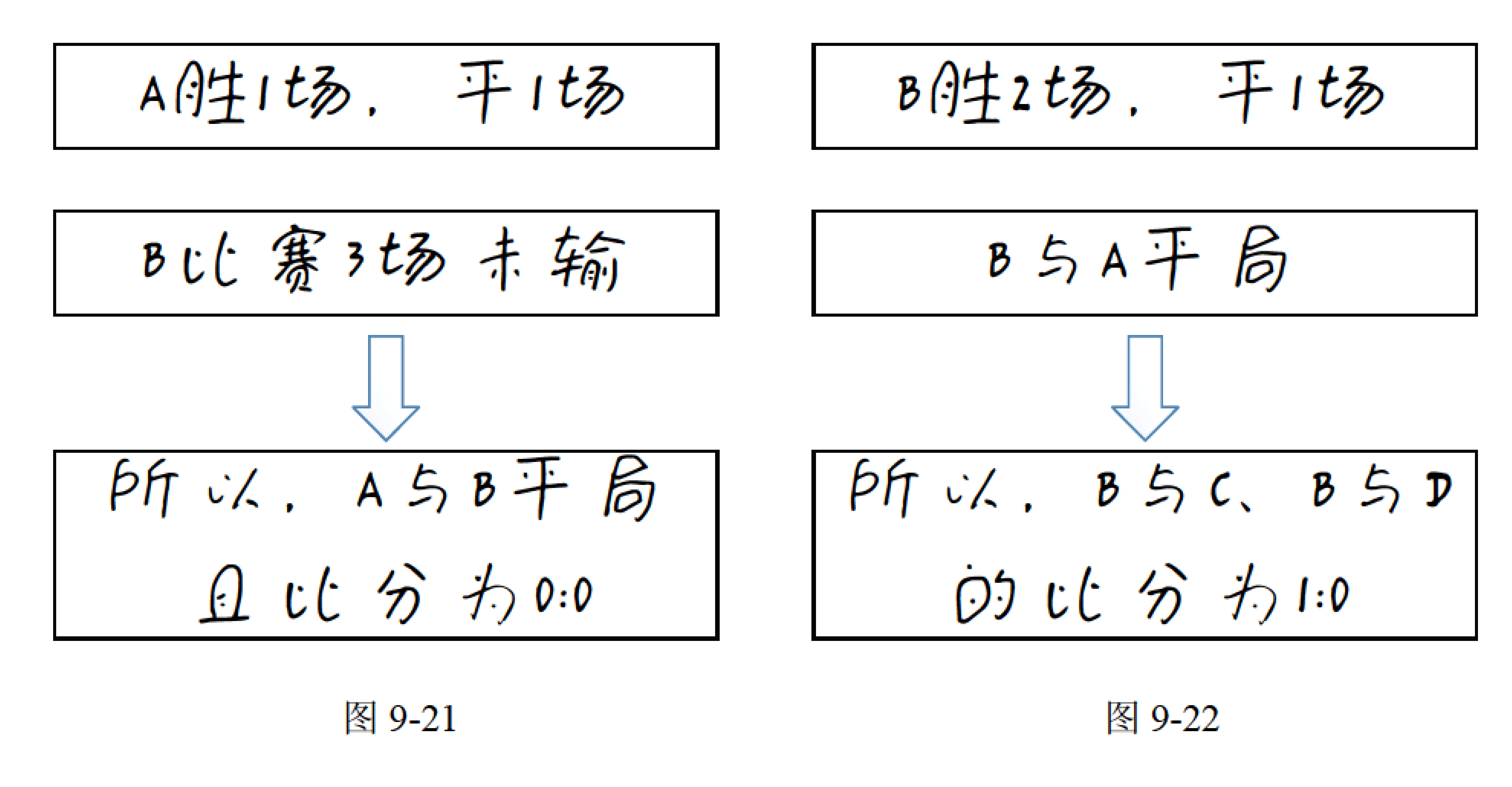


图 9-21　　　图 9-22

（5）根据 A 队进 3 球失 2 球数，且 A 队与 B 队为 0:0 的平局，可知道 A 队另一场比赛的比分应该是 3:2 获胜，如图 9-23 所示。

这时，还只能推理出 A 队获胜局的比分，但不知道是和哪队进行的比赛（可以确定这个比分不是与 B 队的比赛，即 A 队 3:2 获胜的对手可能是 C 队或 D 队）。

（6）再看 C 队（大二队）的成绩，共比赛了 2 场，胜 0 场败 2 场。根据图 9-22 可知 C 队败的一场是与 B 队的比赛（以 1:0 输，失 1 球）。而 C 队共失 5 球进 3 球，可知

另一场失 4 球进 3 球，即以 3:4 的比分输掉比赛，如图 9-24 所示。

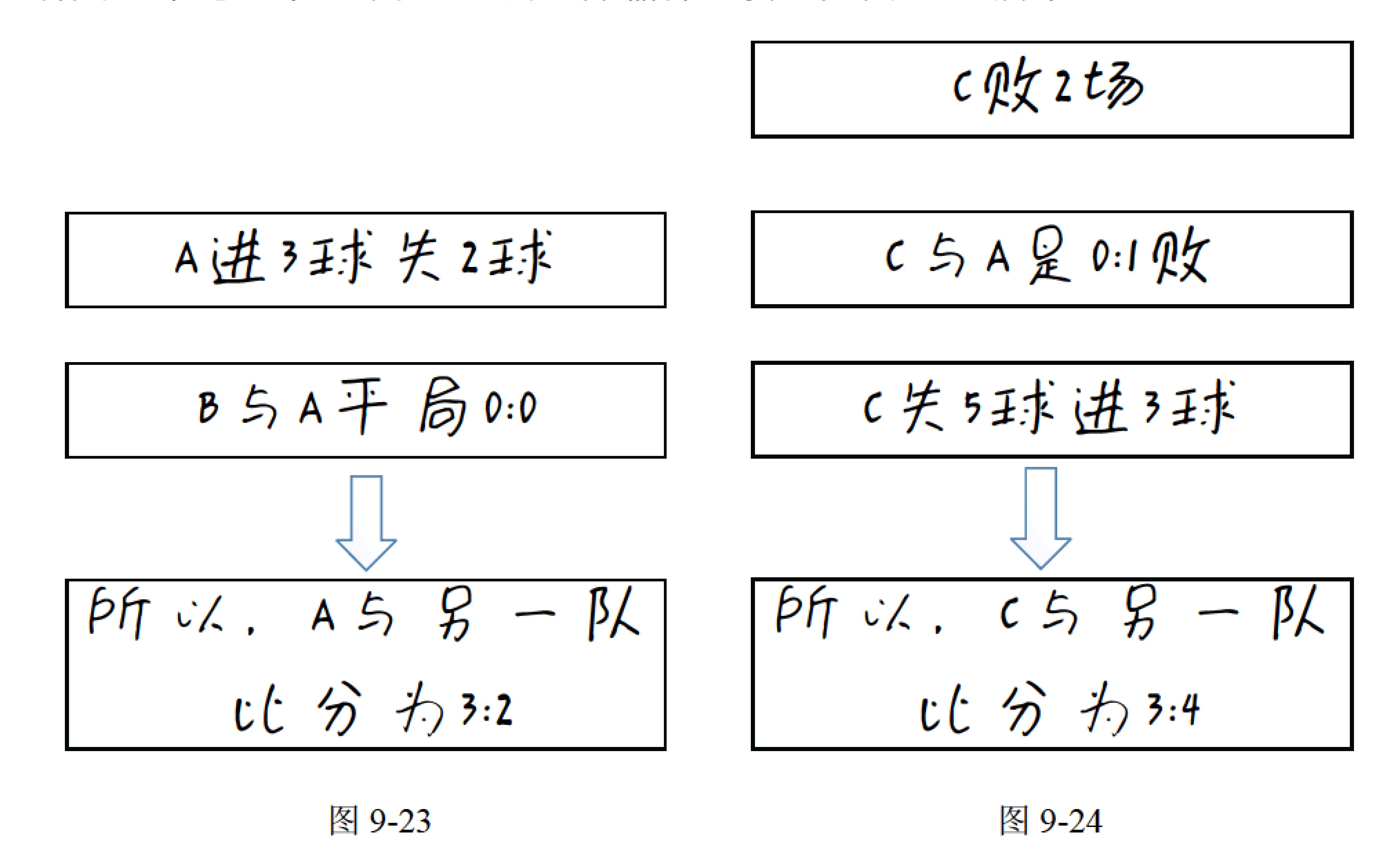


图 9-23　　　　图 9-24

（7）而根据图 9-23，A 队胜利的一场比赛是 3:2，显然，C 队失 4 球的比赛不是与 A 队的比赛，如图 9-25 所示。

（8）C 队不是 3:4 输给 A 队，且 C 队与 B 队的比分是 0:1，一共只与另 3 支球队比赛可知，比分 3:4 应该是 C 队与 D 队的比赛，如图 9-26 所示。

（9）再回过头来看图 9-23，这个图推理出 A 队与另一队的比分为 3:2，而图 9-25 推理出 C 队未与 A 队进行比分为 3:4 的比赛，那么，A 队比分为 3:2 的比赛就只有与 D 队（大一队）进行了。

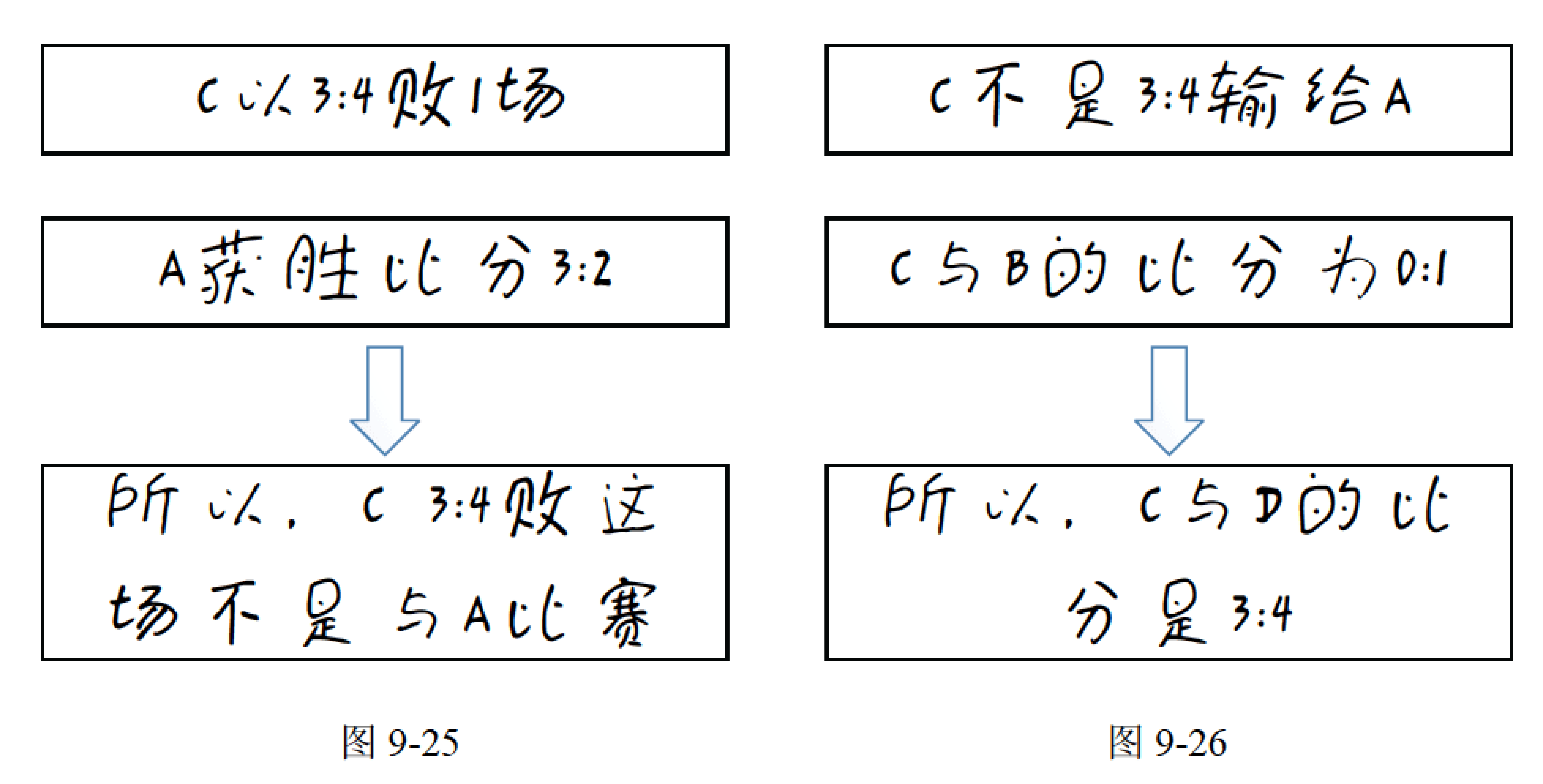


图 9-25　　　　图 9-26

通过上面 9 步推理，可得出 5 局比赛的情况，填入图 9-16 所示的表格，即可恢复数据，得到如图 9-27 所示表格。

比赛球队	比分
大四：大三	0:0
大四：大二	
大四：大一	3:2
大三：大二	1:0
大三：大一	1:0
大二：大一	3:4

图 9-27

根据图 9-27 所示表格中的数据，即可将图 9-18 所示表格最后一行的数据补充完整，得到如图 9-28 所示表格。

球队	已赛场数	获胜场数	失败场数	平局场数	进球数	失球数	积分
大四	2	1	0	1	3	2	4
大三	3	2	0	1	2	0	7
大二	2	0	2	0	3	5	-6
大一	3	1	2	0	6	7	-3

图 9-28

至此，数据恢复工作完成。

第 10 章　几何图形构造

在现实生活中，可见的物体都可以抽象为由点、线、面构成，而这些点、线、面称为几何图形。通过这些几何图形，可帮助人们有效地刻画错综复杂的世界。这一章，我们来研究几何图形构造方面的内容。

10.1　花盆摆放问题

在现实生活中，如果将物体摆放的位置抽象地看作为一个点，则多个物体摆放的位置就可构成几何图形。通过一些奇妙的想法，往往可使物体摆放出现复杂的几何图形。例如，在每年春节时，我们通常可以在公园看到用花摆放出的“春节快乐”字样，如图 10-1 所示。



图 10-1

10.1.1　10 盆花摆成 5 行，每行 4 盆

园艺公司在用鲜花摆放图案或文字时，为了节约成本，总是想用最少的花盆数量摆放出需要的图案。我们下面就来考察这个问题，怎样用较少的数量摆放出需要的图形。

如果要求用 10 盆鲜花摆放出 5 行，每行都必须为 4 盆，应该怎么摆放？

要摆放成 5 行，每行 4 盆的形状，我们大脑中出现的第一个图形应该如图 10-2 所示。

从图 10-2 中可以看出，摆放成规定的样式需要 20 盆鲜花，可现在只给 10 盆鲜花，少一半的数量，该怎么办呢？

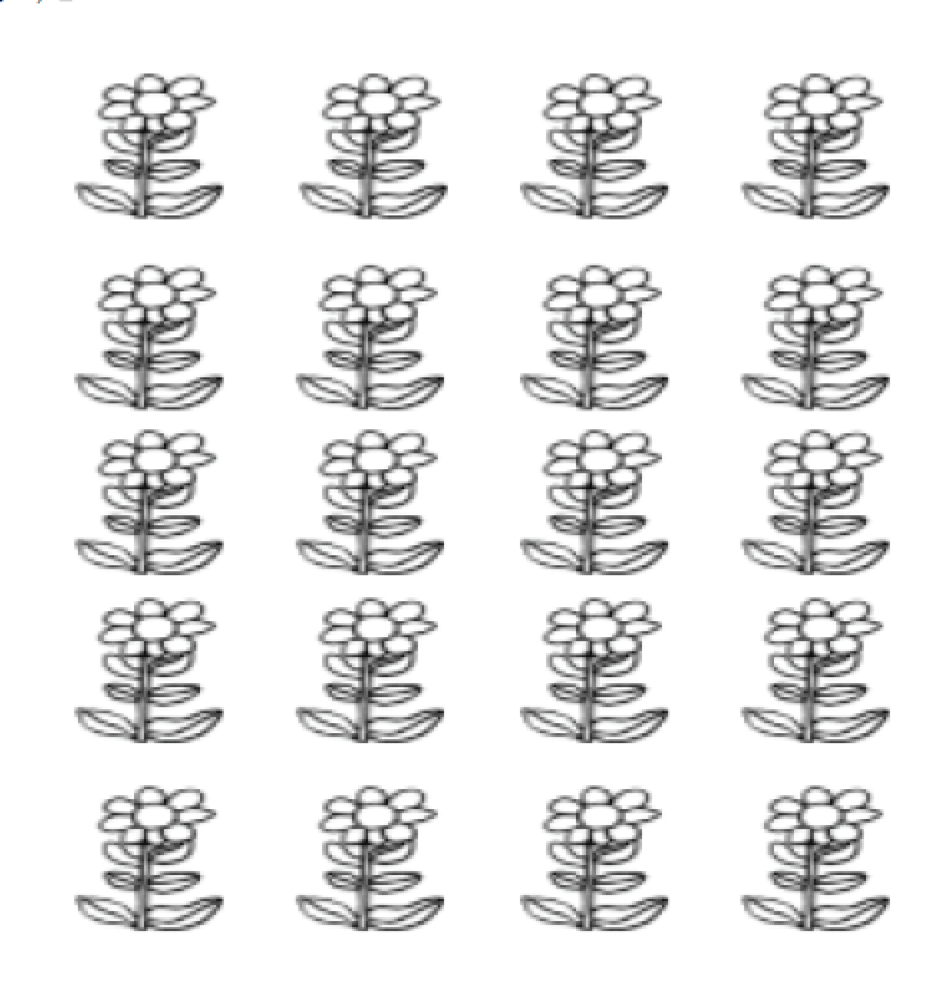


图 10-2

10.1.2　转变思路，找出答案

如果按图 10-2 所示构想来摆放，明显没办法解决问题。因此，这时就需要转换思路了。

分析图 10-2 所示构想图，每行是相互平行的，并且每列也是对齐的，这是最常见的形式。可是，现在并没有要求每行是平行的，也没要求列要对齐。

再接着分析，5 行 4 列，按理论计算需要 20 盆鲜花才够，但现在只有 10 盆鲜花，怎么办？肯定有部分鲜花摆放的位置可以共用。也就是说，有些鲜花既在这一行中，也可看作在另一行中，这样，1 盆鲜花就当作 2 盆用了。例如，如图 10-3 所示，横向看作一行，纵向上也可看作 1 行。这样，每行 4 盆鲜花，2 行只用了 7 盆鲜花（如果两行平行的话，需要用 8 盆）。左图与右图虽在横向位置上不一样，但同样是两行鲜花。

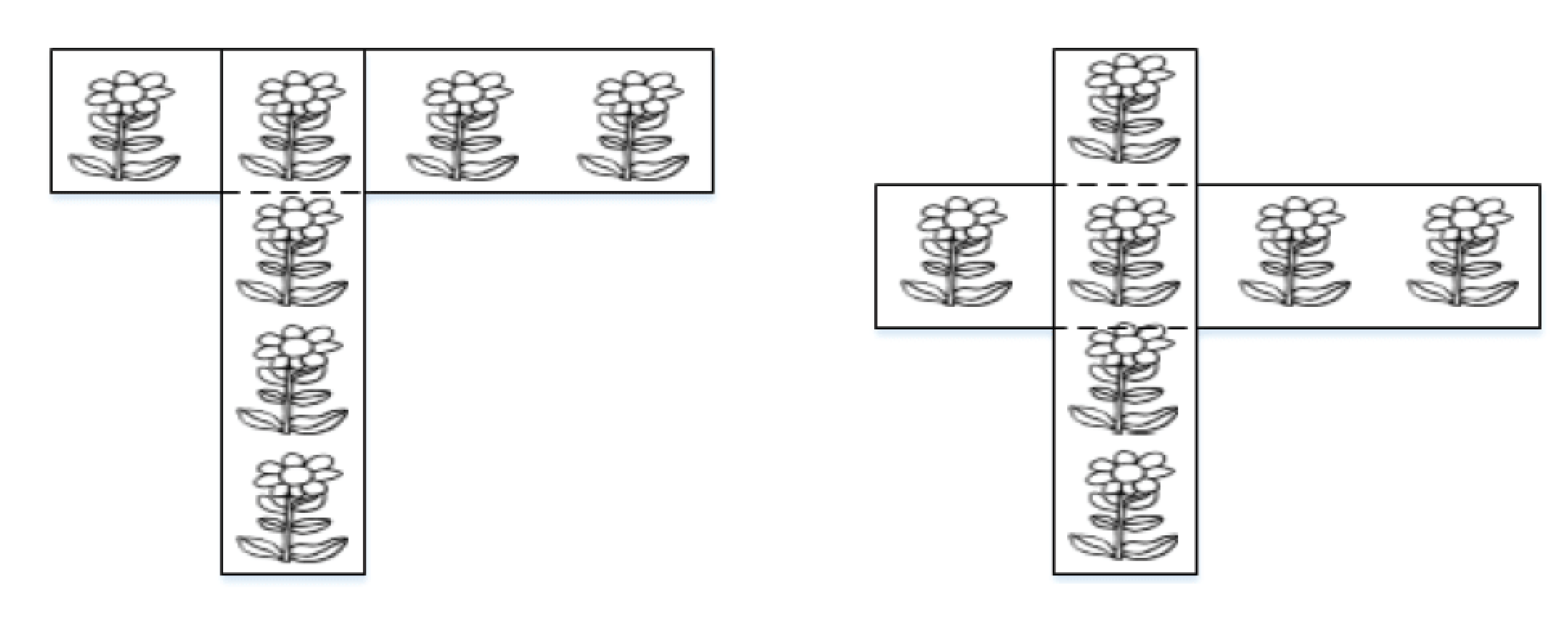


图 10-3

另外，也可斜着排列，如图 10-4 所示。

从图 10-3 和图 10-4 中可以看出，当两行共用某一盆鲜花时，就可减少一盆鲜花的数量。

有了这个思路就好解决问题了。在本例中，每盆鲜花都应该位于两个行中。也就是说，行与行之间有交叉。

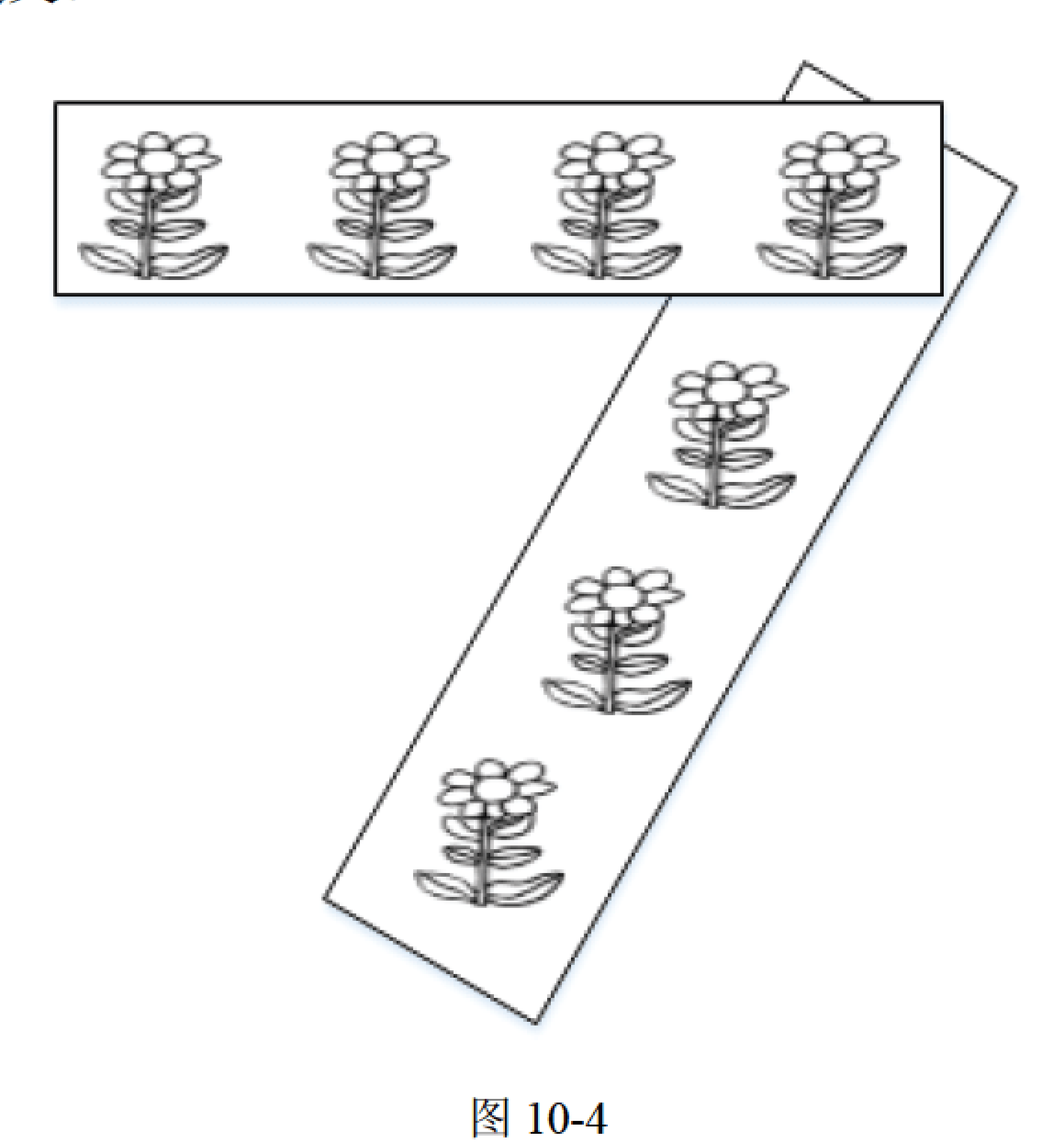


图 10-4

因此，可以得出按图 10-5 所示图形摆放鲜花的方式。

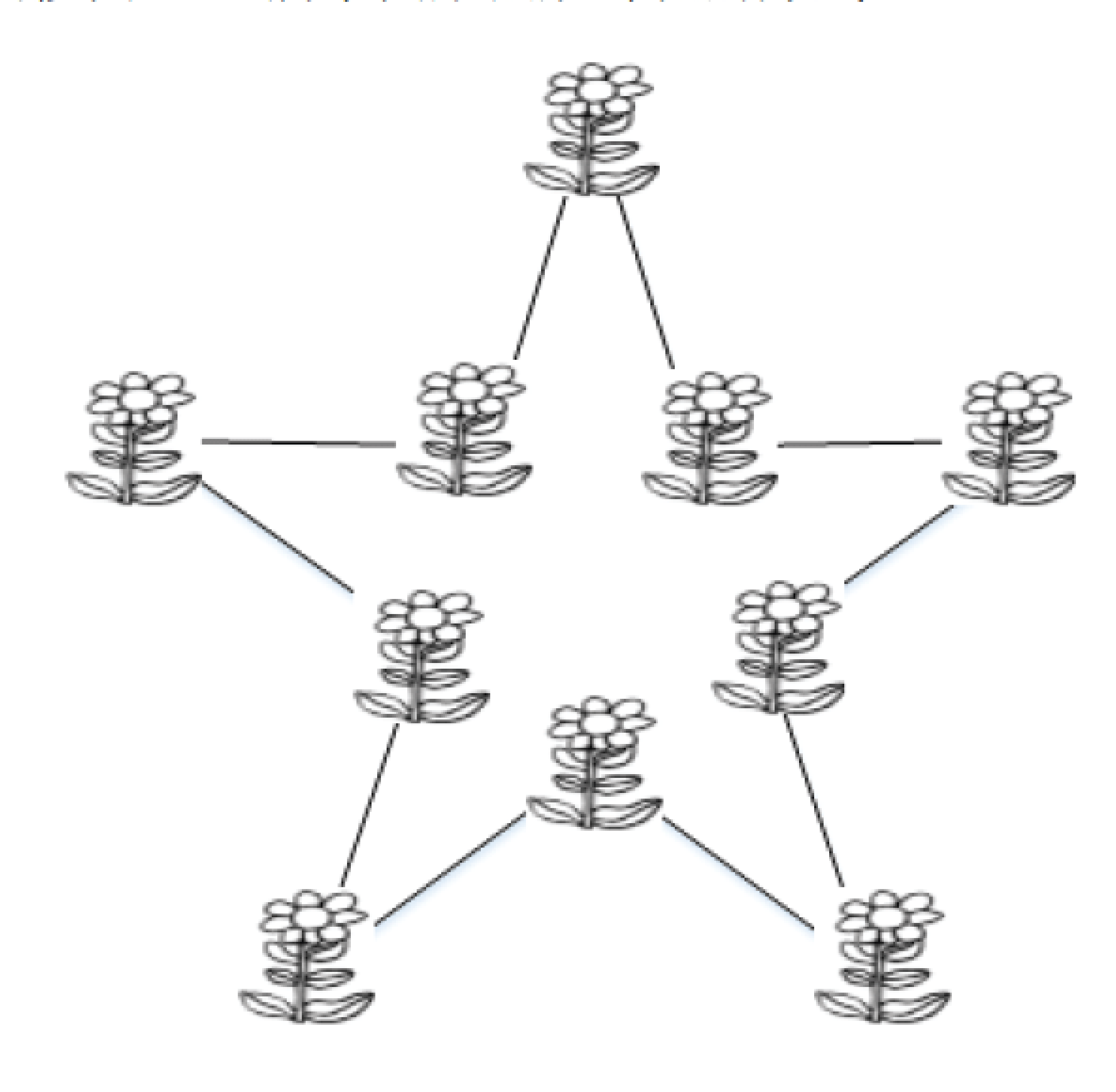


图 10-5

图 10-5 中将鲜花摆放成了一个五角星，在五角星的每一条边上都有 4 盆鲜花，并且

五角星有 5 边，也就是有 5 行。可以看出，每盆鲜花都位于两行之中。这样，只用 10 盆鲜花就摆出了 5 行，每行 4 盆的效果。

10.1.3 升级问题（10 盆花摆 10 行，每行 3 盆）

有了以上解决方法，现在我们将问题升级：如果这 10 盆鲜花要摆成 10 行，每行为 3 盆，该怎么摆放?

每行 3 盆，共 10 行，如果每盆鲜花只在一行上，则一共需要 30 盆鲜花才够。现在只有 10 盆，那么，每盆鲜花需要位于 3 行中。

因此，可设计出如图 10-6 所示的摆放形式。

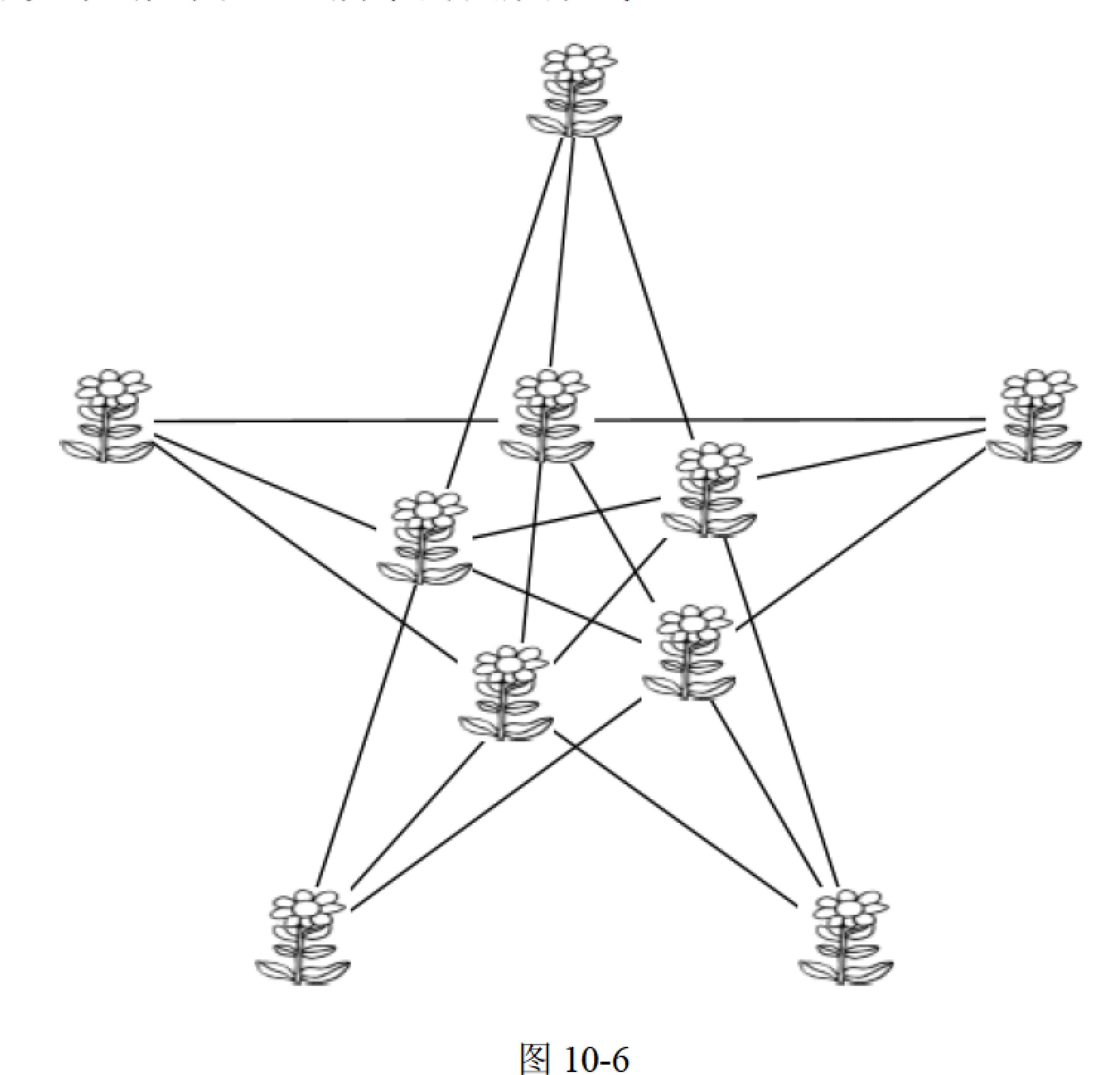


图 10-6

在图 10-6 所示图形中，仍然是以五角星为基础。五角星每个角由两条线相交构成，但是，我们现在需要每盆花要在 3 行中，因此，就需要从五角星的角出发，再绘制一条直线。图 10-6 中将新绘制的线段与五角星的另两边相交，这样，新绘制的线段也就至少有 3 个交点了。也就是说，从一个角出发就有 3 条线段，就可以摆放 3 行鲜花了。每个角都有 3 条线段相交，并且，新绘制的线段也和其他角绘制的线段相交，最终就可得到图 10-6 所示图形。

细看图 10-6，其实是在一个大的五角星的 5 个角上摆放 5 盆鲜花，接着在内部还有一个较小的五角星，也分别在 5 个角上摆放 5 盆鲜花。这样，就摆放完 10 盆鲜花，并使每行有 3 盆鲜花。

10.2　残缺的棋盘能补上吗？

棋盘是我们常见的围棋、象棋、国际象棋等棋类游戏中不可或缺的基本道具，在棋盘中隐藏着许多有趣的图形构造问题。这节我们来做一个有趣的棋盘图形构造题。

10.2.1　被切割的棋盘

我们来看一个著名的问题，这个问题曾作为中国科技大学少年班的招生试题。如图 10-7 所示是一个残缺的国际象棋棋盘，在棋盘左上角和右下角都有一个方格被剪切了。请问，能不能用 31 个如图 10-8 所示的 2×1 的矩形拼出图 10-7 所示的残缺棋盘。

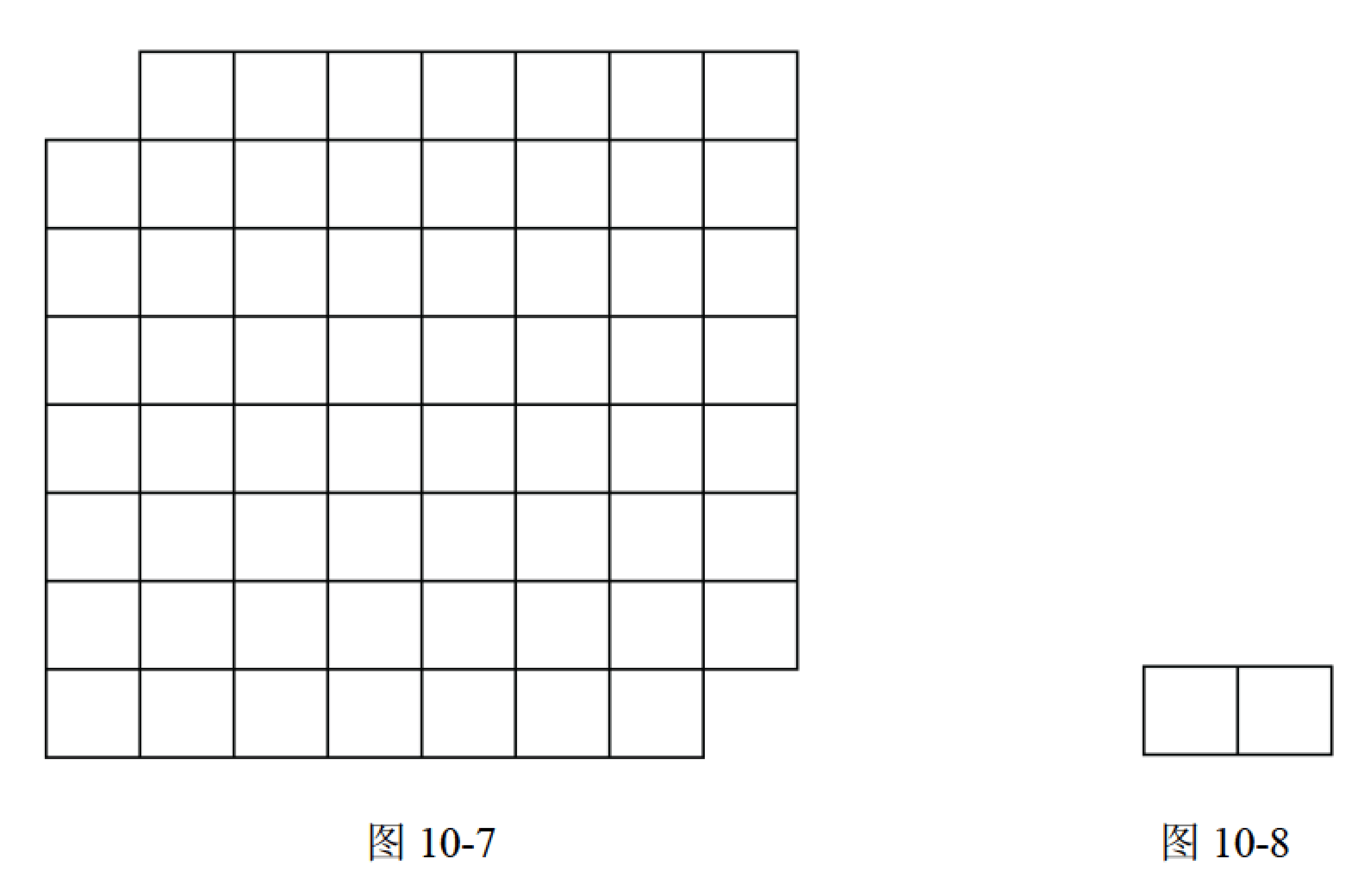
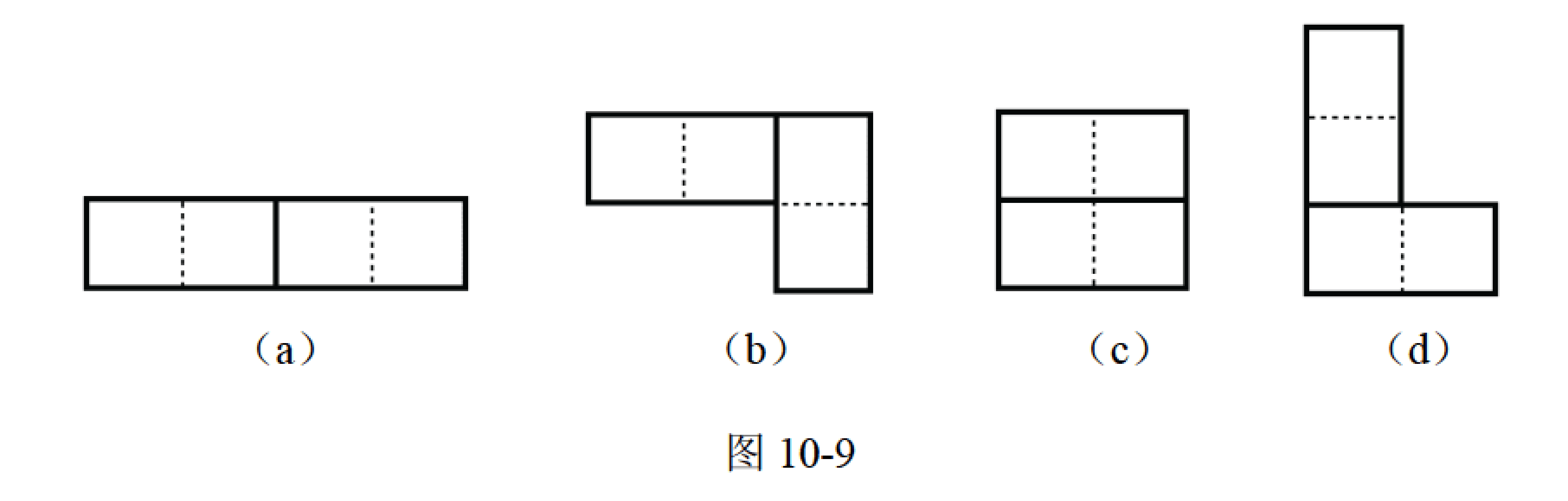


图 10-7　　　　图 10-8

图 10-8 所示为棋盘的 2 格，在拼接时可以旋转，但不能重叠。例如，可拼接成图 10-9 所示的图形（为了方便辨识图 10-8 所示长方形矩形，在该长方形矩形中两个方格之间用虚线分隔标识），其中图（a）是将两个矩形放在同一行，图（b）左侧是横着放置的矩形，右侧将矩形进行了旋转，图（c）将两个矩形放在同一列，图（d）上方将矩形进行了旋转，下方横着放置矩形。



（a）　（b）　（c）　（d）

图 10-9

图 10-9 所示两个矩形之间没有重叠，是可以的。但是，不能用图 10-10 所示的方式

进行拼接，因为这时两个矩形有一部分重叠了。图（a）看起来是三个正方形块了，其实是将两个矩形的一半重叠在了一起；同样，图（b）也是这种情况，只是将矩形旋转成竖直放置了；图（c）则是一个水平放置的矩形与一个垂直放置的矩形进行了重叠。

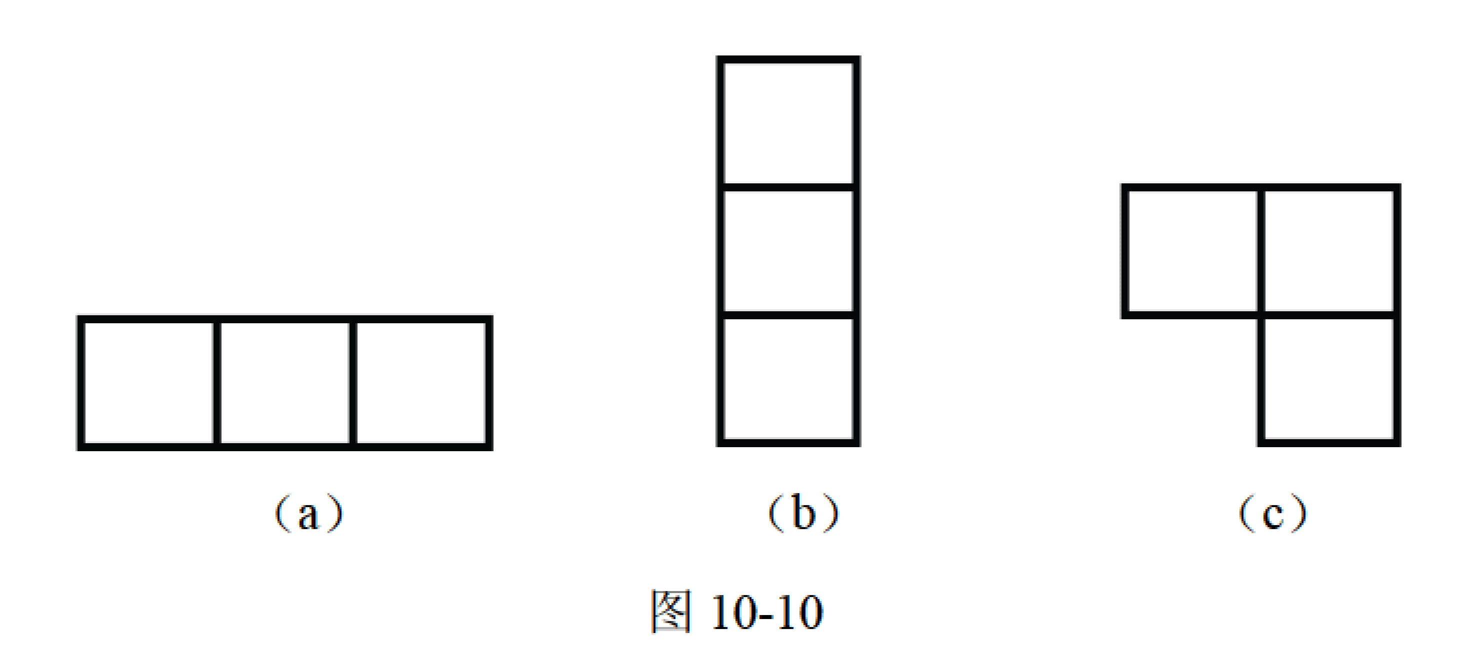


图 10-10

10.2.2　能拼接出残缺棋盘吗

先对图 10-7 所示残缺棋盘进行计算，完整的棋盘共有 8 行 8 列：

$$8 \times 8 = 64$$

即应有 64 个正方形的方格。

根据题意，左上角和右下角各剪掉一个方格，则只有 62 个方格。而图 10-8 所示矩形是由两个方格组成的，因此：

$$62 \div 2 = 31$$

正好是 31。

因此，很容易得出能用 31 个图 10-8 所示矩形拼接出图 10-7 所示残缺棋盘的结论。

可是，这是错误的！

错！　不能！

为什么是错误的呢？我们来验证一下。如图 10-11 所示，从左上角开始放置图 10-8 所示长方形矩形，第 1、2、3 个矩形都进行水平拼接，第 4 个矩形旋转 90° 后拼接。接着，在第 4 个矩形的左边、第 3 个矩形的下方拼接第 5 个矩形。然后向左方拼接第 6、7 个矩形，第 8 个矩形又旋转 90°，第 9 个矩形又开始向右拼接……，这样循环进行，直到第 27 个都还能正常拼接，但是，准备在右下角拼接第 28 个时发现只剩一个方格（右下角带斜线的方格），没办法拼接了！

因此，可知道前面的结论是错的！不能用 31 个图 10-8 所示矩形拼接出图 10-7 所示

残缺棋盘！

图 10-11

采用图 10-11 所示的方法逐个进行拼接，最终得出一个否案的答案。可是这种方法比较麻烦，要逐个去测试。

其实，我们还可以有更快速的方法。

仔细分析，国际象棋棋盘是用颜色进行了标识的，棋盘格采用黑、白相间涂色，如图 10-12 所示。从图中可看到在一个白格的上、下、左、右都是黑格，同样，在一个黑格的上、下、左、右都是白格。因此，图 10-8 所示由两个方格组成的矩形一次能且只能拼接棋盘中的一个黑格和一个白格。

如图 10-12 所示，残缺棋盘中共有 62 个方格，其中有黑格 32 个，白格 30 个。根据前面的分析，用图 10-8 所示的矩形进行拼接时，这个矩形的两个方格中有一个黑格一个白格。那么，31 个矩形拼接成的图形应该有 31 个黑格、31 个白格。而图 10-12 中却为 32 个黑格、30 个白格。因此，没办法用图 10-8 所示矩形拼接出图 10-7 所示的残缺棋盘。

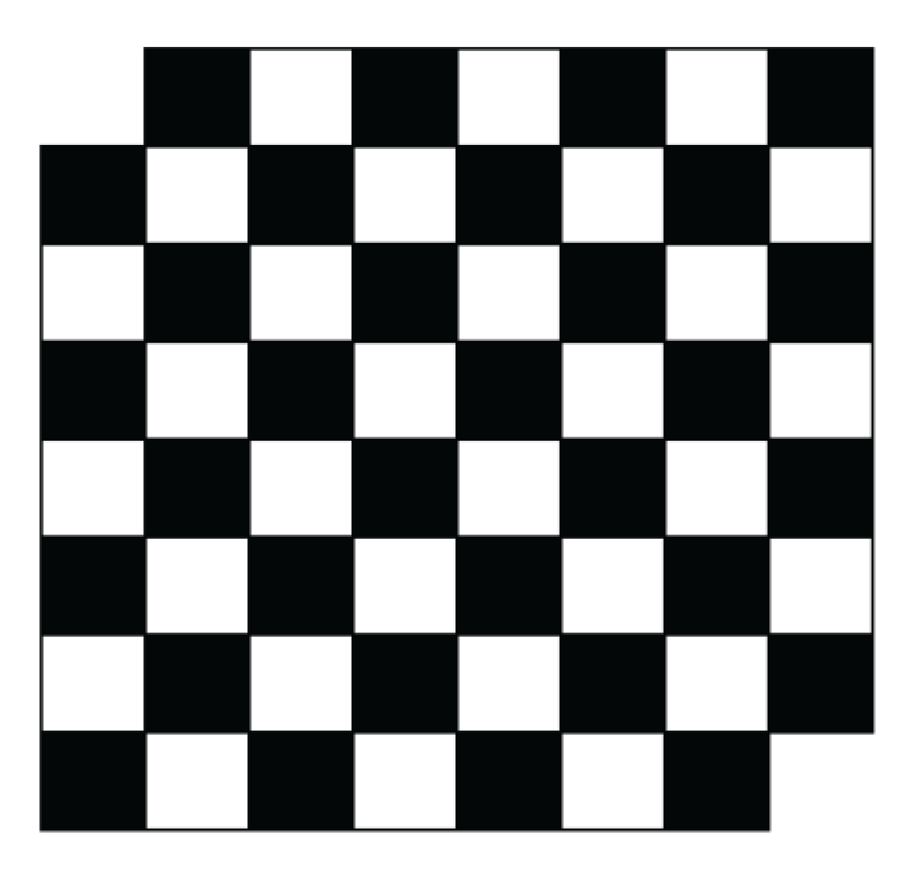


图 10-12

在这个例子中，看似简单的几何图形构造，如果不经推敲分析，很容易得出错误的结论。由此可见几何图形构造的有趣之处。

10.3　线条哪里去了？

10.2 节介绍的是方格的判断，这节我们来看一个与线条有关的有趣问题。让我们从一个魔术开始。

10.3.1　神奇的魔术

魔术师拿出一张白纸展示给观众看，然后在这张纸上画了 10 根线条，并将画线后的纸张展示给观众，逐一数了一下线条的数量，没错，是 10 条，如图 10-13 所示。

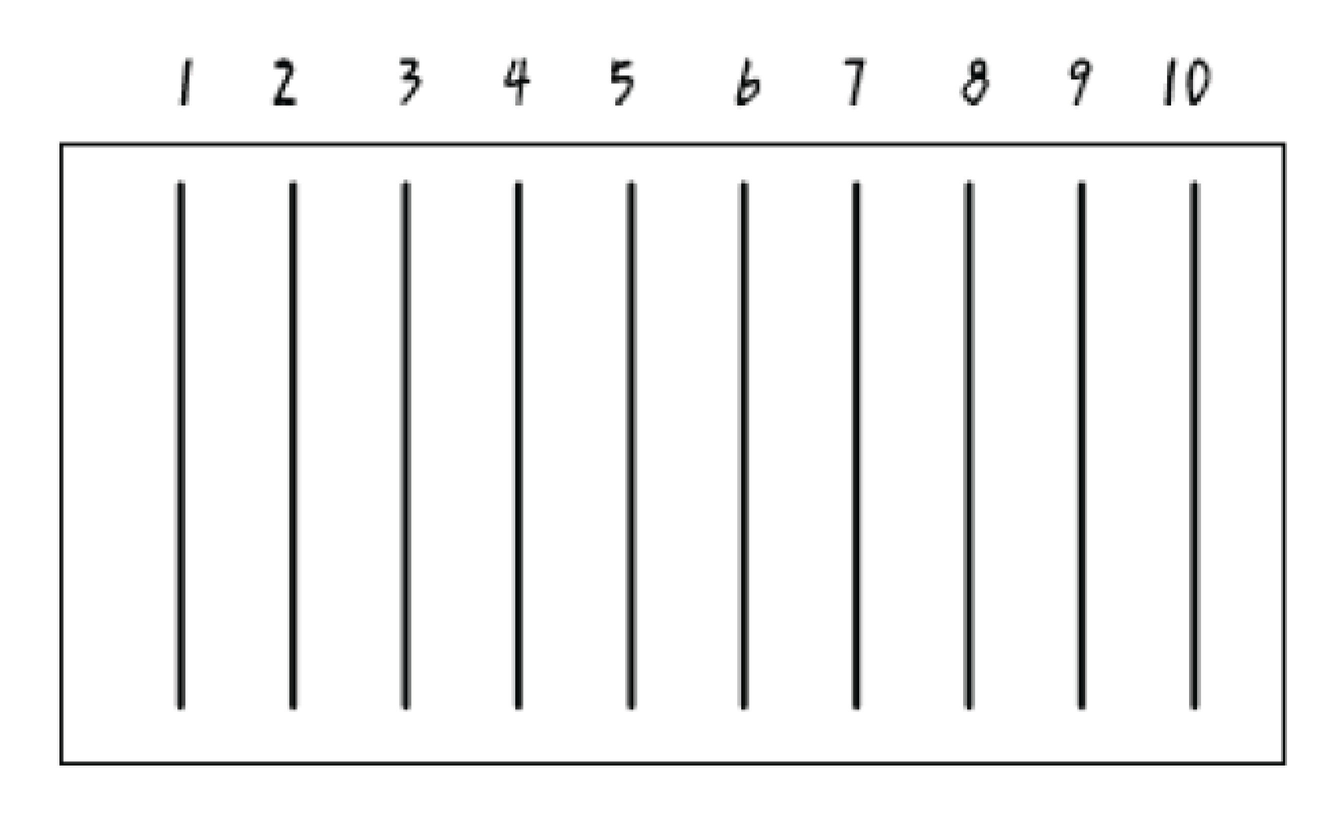


图 10-13

接着，魔术师拿起剪刀沿纸张对角剪了下去，剪完以后，魔术师又将剪成三角形的两部分拼接起来。这时，展示给观众看时，这张纸上只有 9 根线条了！如图 10-14 所示。

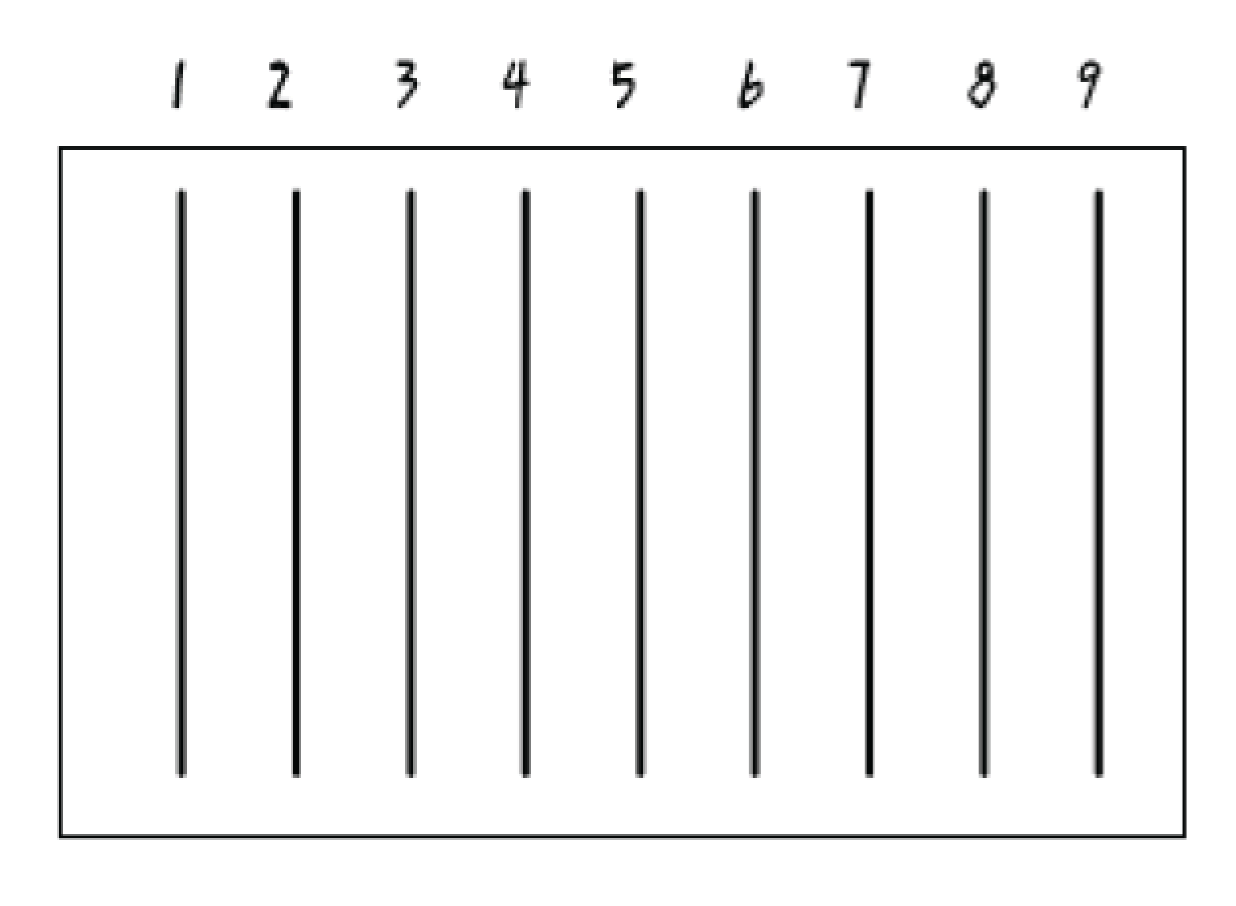


图 10-14

魔术师并没有将某一根线条剪下扔掉！只是沿对角剪了一下，没有扔掉任何东西，还是原来的纸张，原来 10 根线条，怎么少了一根？这根线条到哪里去了呢？

10.3.2　解析丢失的线条

为什么会丢失一根线条呢？其实原理很简单，下面我们来分析一下。

首先，在画 10 条线条时，每条线之间的间距要相等，线条等长，并且在高度方向上对齐。

接着，连接第 1 条线的下端与第 10 条线的上端画一条虚线，如图 10-15 所示。沿着这条虚线将纸张剪开，就得到如图 10-16 所示的两张三角形的纸张。可看到，每张三角形纸张中都只有 9 条线，左图中第 10 条线没有，该条线完全位于右图中，同样，右图中第 1 条线也没有，该条线完全位于左图中。

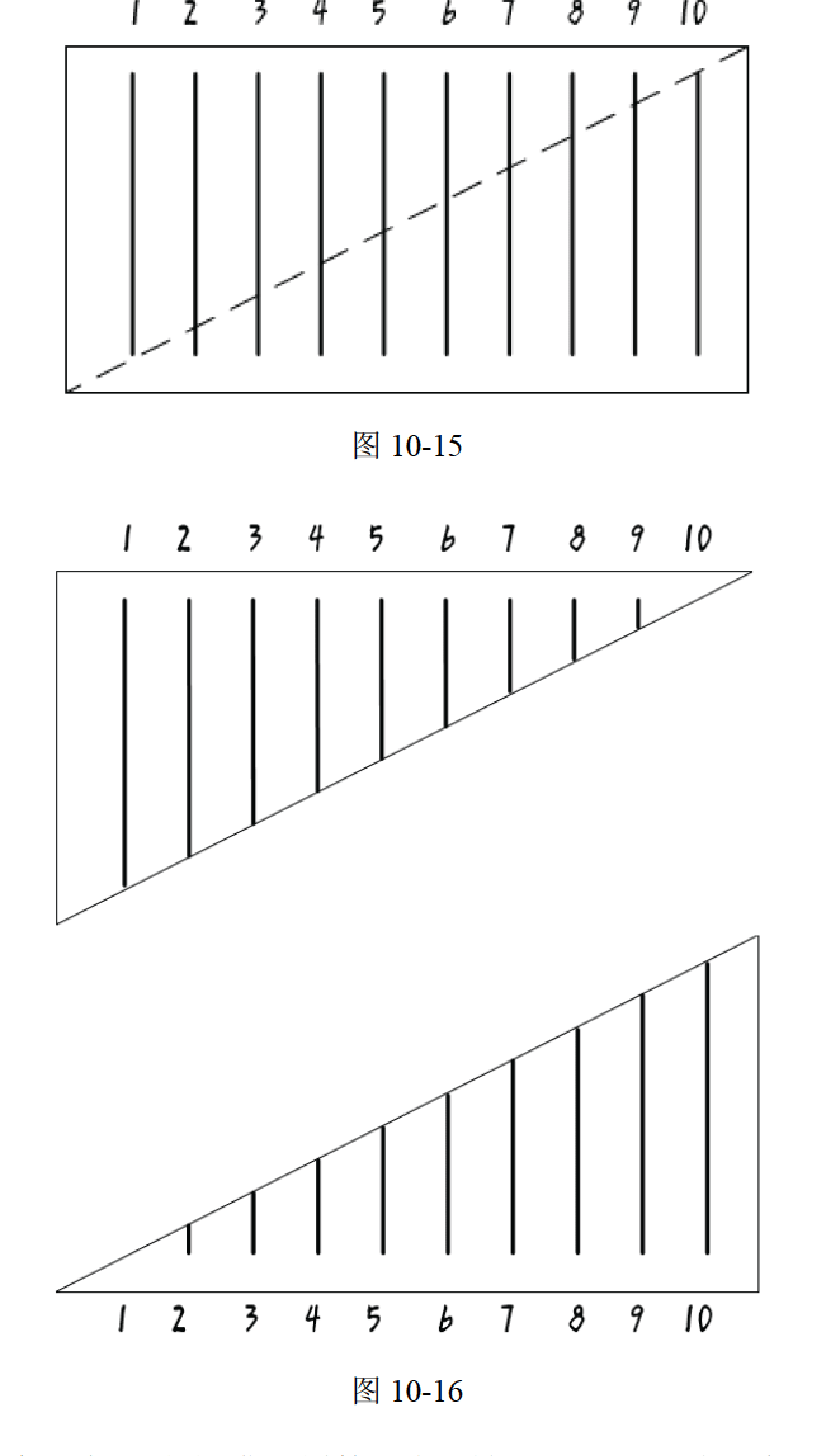


图 10-15

图 10-16

接下来，将两个三角形进行错位拼接，得到如图 10-17 所示结果。数一下图 10-17

拼接后的线条数量，只有 9 条！

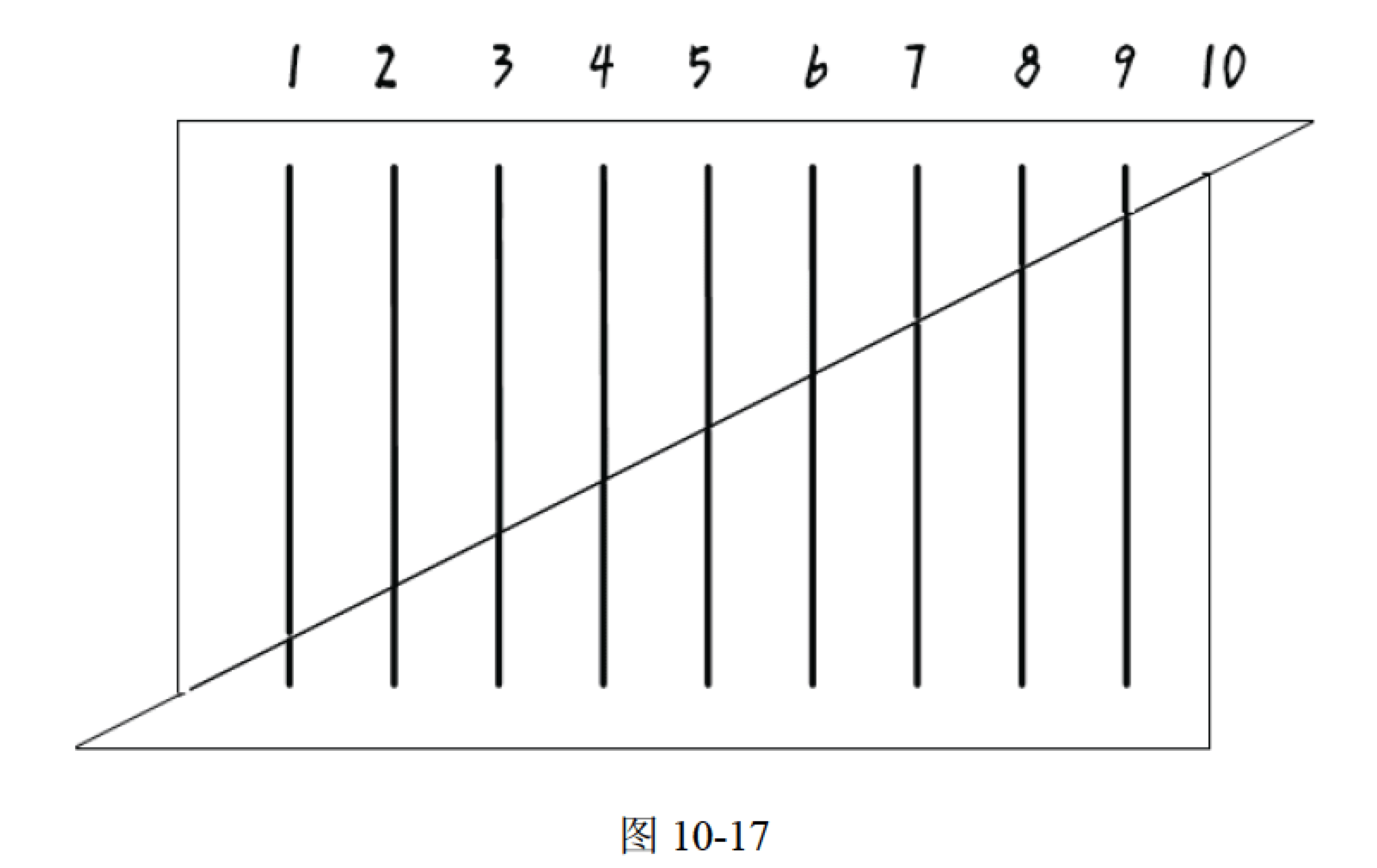


图 10-17

从图 10-16 中可看出，当魔术师将纸张剪成两个三角形时，实际上是将原来的 10 条线条剪分成了 18 条长短不一的线条（每个三角形中有 9 条）。由于最初画线时 9 条线的间距是相同的，因此，在重新拼接时，可进行错位拼接，就可得到最终的 9 条线条。

在图 10-17 中可看到，原来的第 1 条线下方增加了一部分长度，增长了多少呢？我们来计算一下，设最初画的线条长度为 x，则 10 条线长度共为 $10x$。而图 10-17 中共有 9 条线，每条也相等，设每条长度为 y，则 9 条线长度共为 $9y$。显然，图 10-17 所示 9 条线的总长度与图 10-13 所示 10 条线的总长度是相等的，也就是说：

$$10x = 9y$$

则：

$$y = \frac{10x}{9}$$

也就是说，拼接后每条线的长度比初始长度要长$\frac{1}{9}$。

类似地，如果将图 10-17 所示下方三角形向右移动，又可恢复到 10 条线条的状况。

10.4 图形剪拼

通过前面几个例子可看出，即使是很简单的几何图形构造，也可演化为一个有趣的

问题。在本节中，我们再来看几个有关图形剪切、拼接的例子。

要把一个几何图形剪成几块形状相同的图形，或是把一个几何图形剪开后拼成另一种满足某种条件的图形，完成这样的图形剪拼，需要考虑图形剪开后各部分的形状、大小及它们之间的位置关系。

10.4.1　均分三角形

如何将一个等边三角形均分为 8 个形状、大小都一样的三角形？

首先可以想到的是，在等边三角形的三边分别取中点，然后连接这些边的中点，得到如图 10-18 所示的图形。

经过一次分割，在图 10-18 所示图形中得到了 4 个形状、大小都相同的三角形，并且这 4 个三角形都是等边三角形。

接下来就好办了，要得到 8 个形状、大小都相同的三角形，只需要将图 10-18 所示的 4 个等边三角形分别切分为 2 个对称的三角形。即将 4 个三角形中的每一个都通过一个角作一条平分线，如图 10-19 所示，就得到了 8 个形状、大小都相同的三角形了。

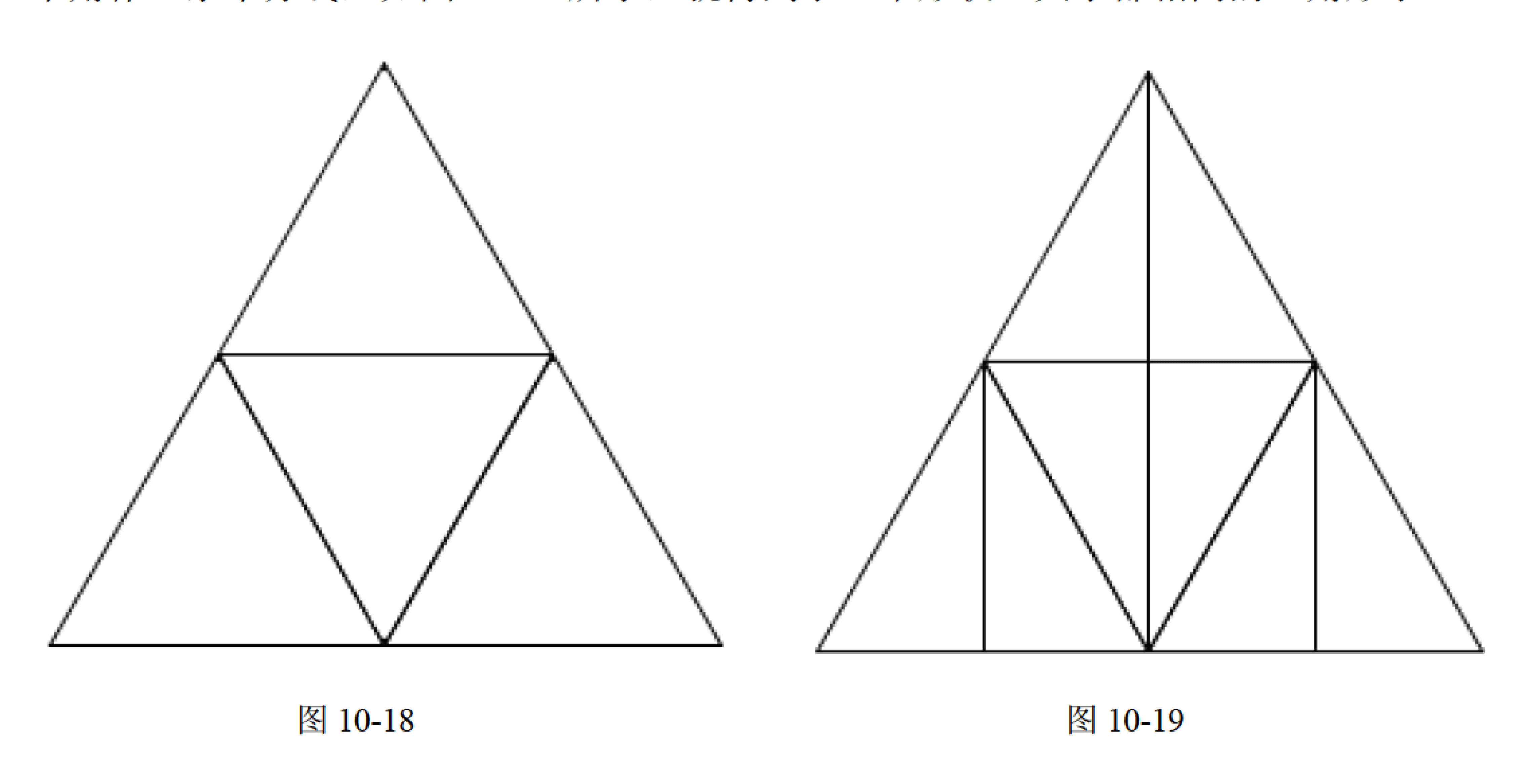


图 10-18　　　　图 10-19

当然，也可以做出与图 10-19 所示不同的图形，只要从图 10-18 所示的 4 个三角形的任一角作平分线，都可将一个等边三角形平分为形状、大小相同的两个三角形。

对要求进行一下变化，如果要将一个等边三角形均分为 9 个形状、大小都一样的三角形，该怎么分割？

其实，要将等边三角形均分为 9 份，可以先将每条边平均分为三等份（图 10-18 中是将边均分为两等份），然后再把分点彼此连接起来，就可得到如图 10-20 所示的图形了。从图中可看到，这个等边三角形已被分割成 9 个形状、大小完全相同的三角形了。

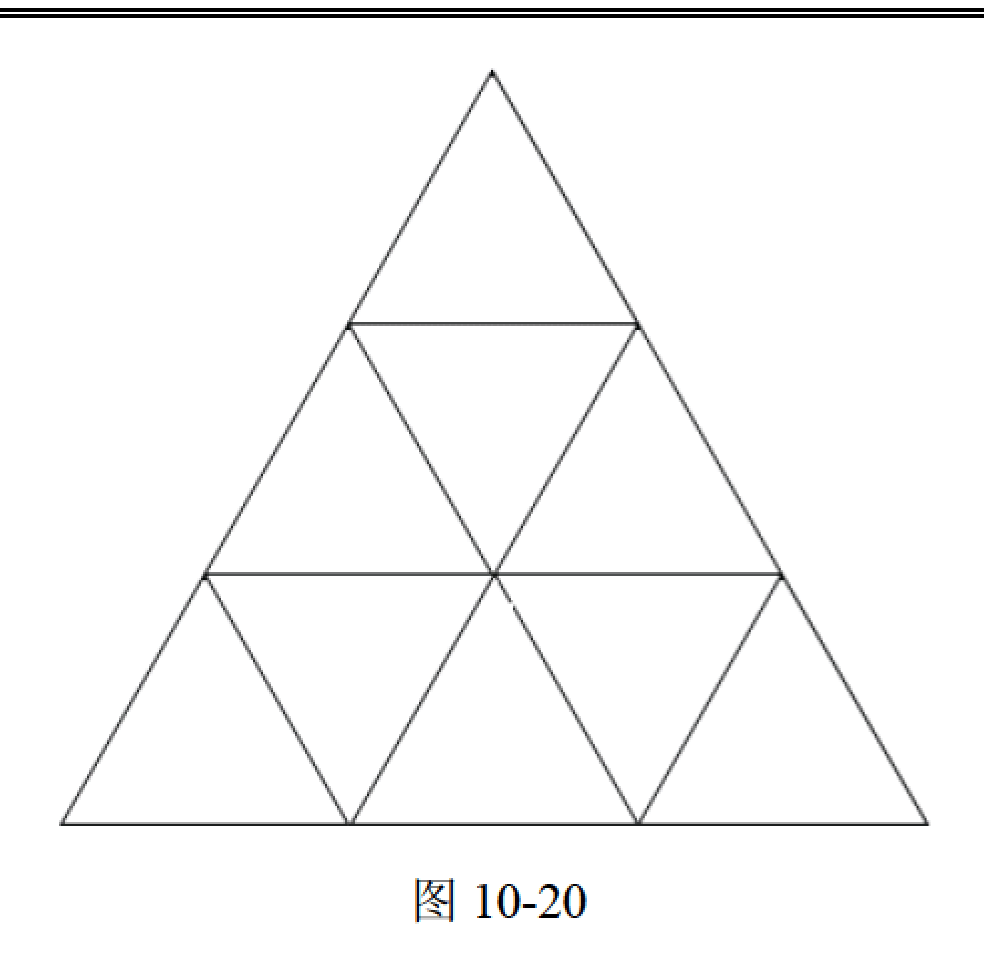


图 10-20

可以看出，对于等边三角形的分割比较简单，下面我们再来看看对矩形图形的分割。

10.4.2 拼接正方形

如图 10-21 所示是一个由多个正方形块组成的平面图形，其中，左上角缺少了 4 个方格。现在能不能将该图分割成 3 块，然后再拼接成一个正方形？

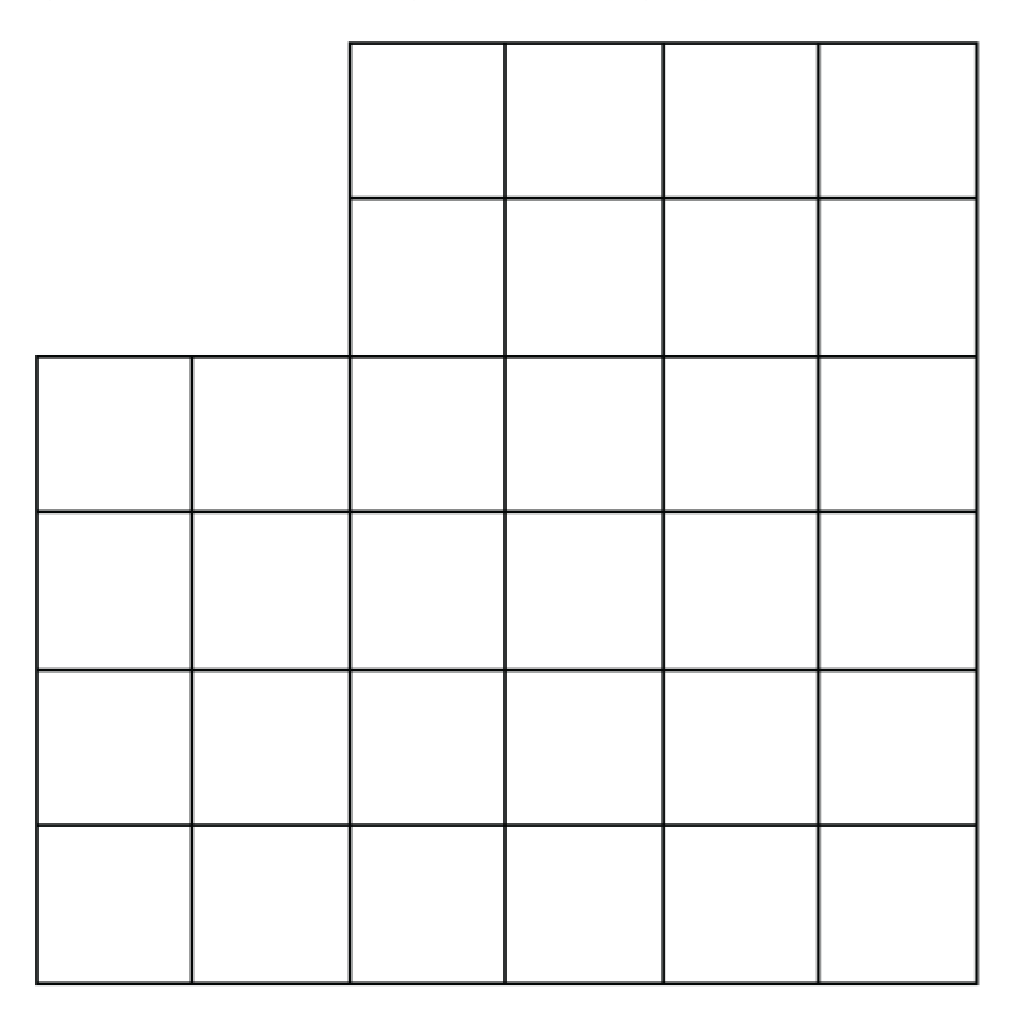


图 10-21

设每一个小方格的边长为 1。

首先对图 10-21 进行分析。这是一个残缺的正方形，边长为 6，其面积应该为 36。但是，由于左上角缺了 4 个方格，因此该图的面积为 32。因此，最后拼接成的正方形的面积也应该是 32，才能完成题目的要求。

原图形面积是 32，所以拼成正方形的面积也应是 32，即正方形边长应为 $\sqrt{32}$，不是一个整数。如果正方形的边长不为一个整数，对图 10-21 的图形的分割就比较麻烦，感觉不好着手。

我们还是从最后拼接成的正方形的边长入手，根据前面的计算其边长应该为 $\sqrt{32}$，即 $4\sqrt{2}$。图 10-21 所示图形是由若干个边长为 1 的小正方形组成的，而边长为 1 的小方

格的对角线长为$\sqrt{2}$，如图 10-22 所示。

如果要使拼接后的正方形的边长为$4\sqrt{2}$，我们可以考虑用 4 个小方格的对角线连线来作为边长。

有了这个思路，就可考虑将图 10-21 作如图 10-23 所示的分割线了，图中从左上方向右下方的虚线 AB 就是分割线。线段 AB 的长就是$4\sqrt{2}$。

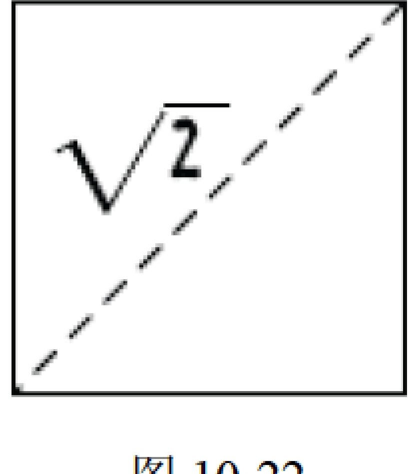


图 10-22

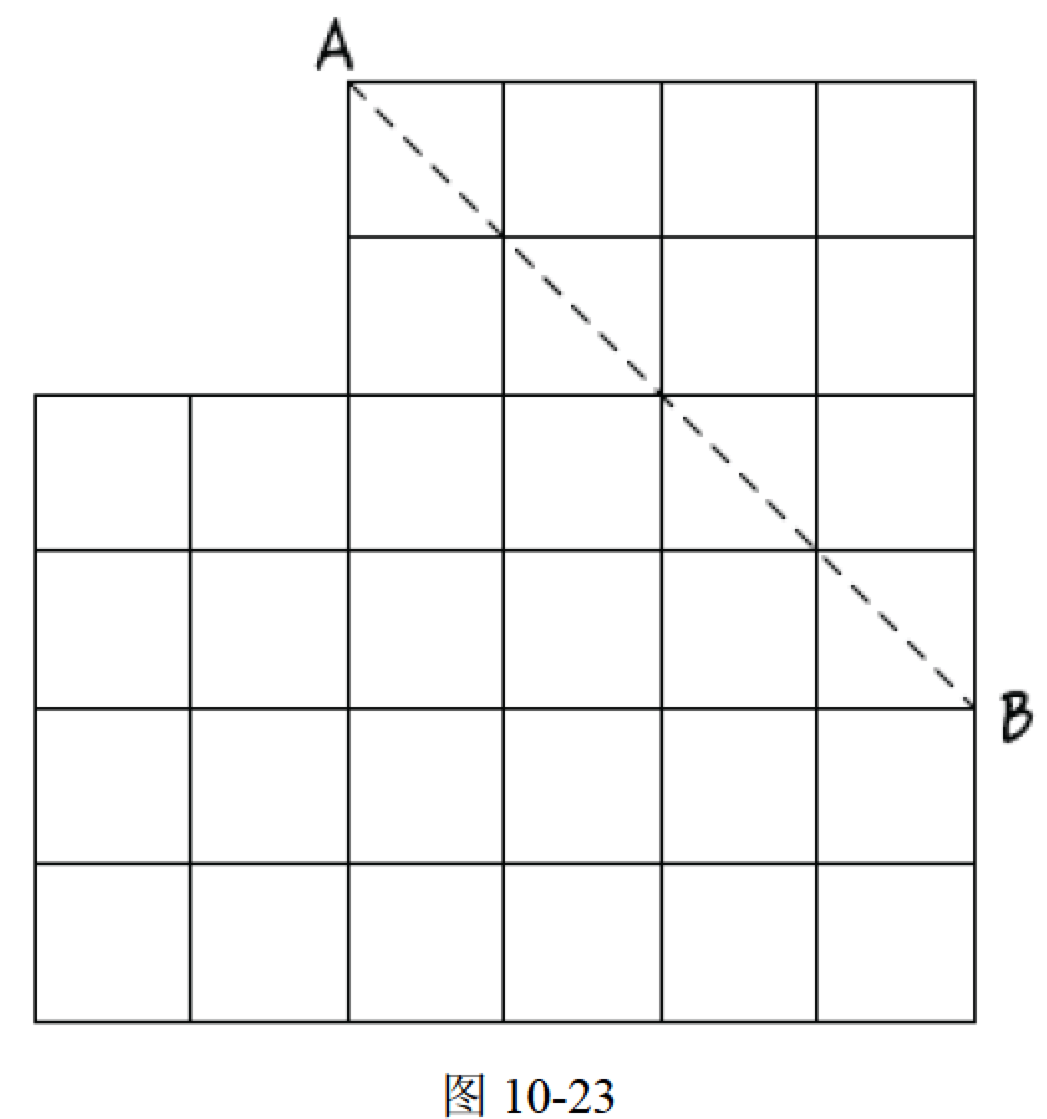


图 10-23

这样，我们就找到拼接正方形的两条边了（分割以后就是两条边长为$4\sqrt{2}$的边）。

接下来还需要分割出正方形的另两条边。在图 10-23 所示分割图形中，虚线 AB 右上方是一个规则的图形，并且最长边就是 AB，再次分割的可能性小。而左下方的图形不规则，且面积较大，还可从多个角度找到分割边长为$4\sqrt{2}$的组合。例如，制作一条与 AB 平行的、从左上角到右下角的分割线，得到如图 10-24 所示分割结果（为了辨识方便，这里将图形的角部分别添加上一些字母标识）。

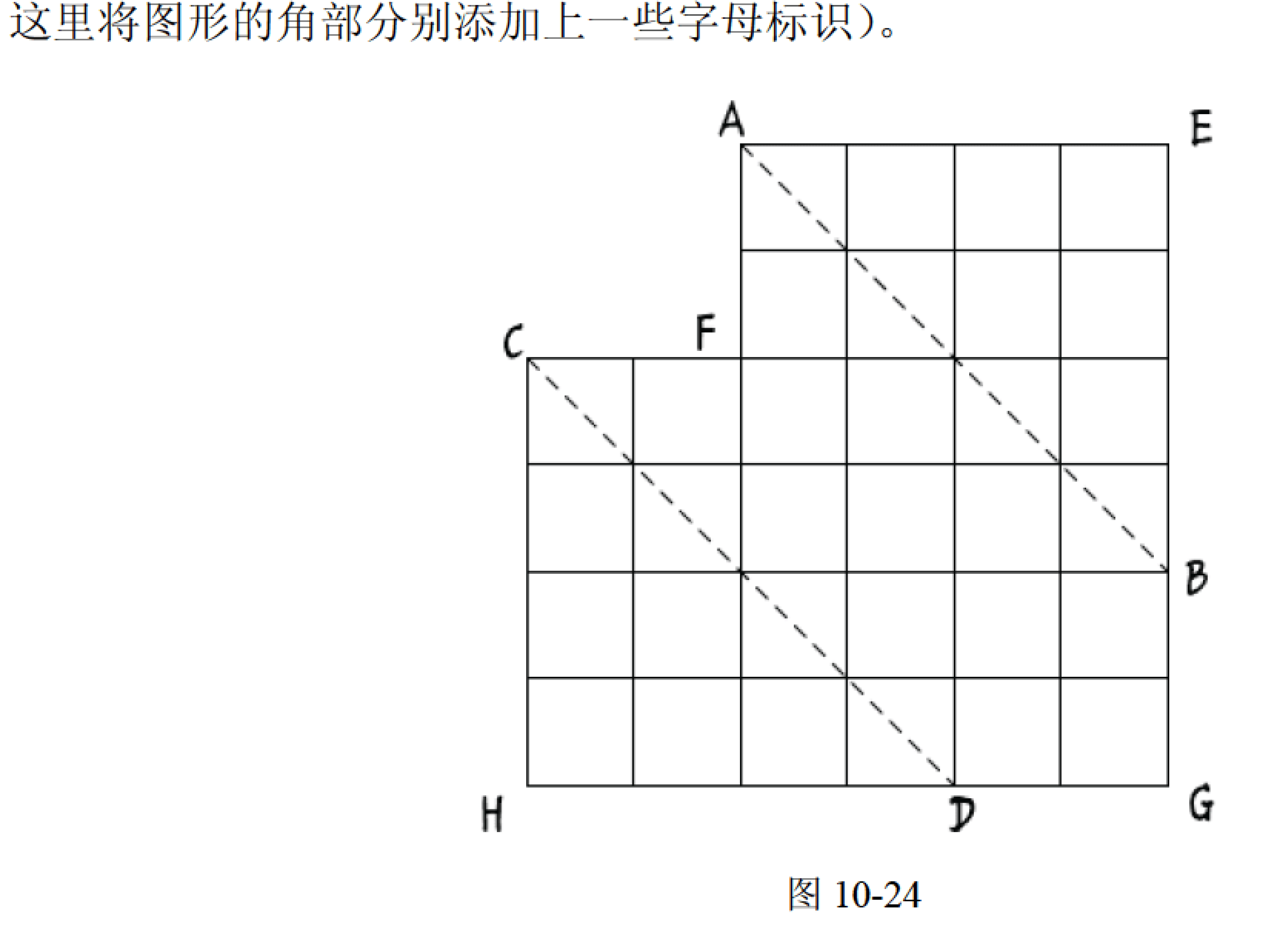


图 10-24

将图 10-24 通过虚线分割的三部分分离出来，得到如图 10-25 所示的三部分。

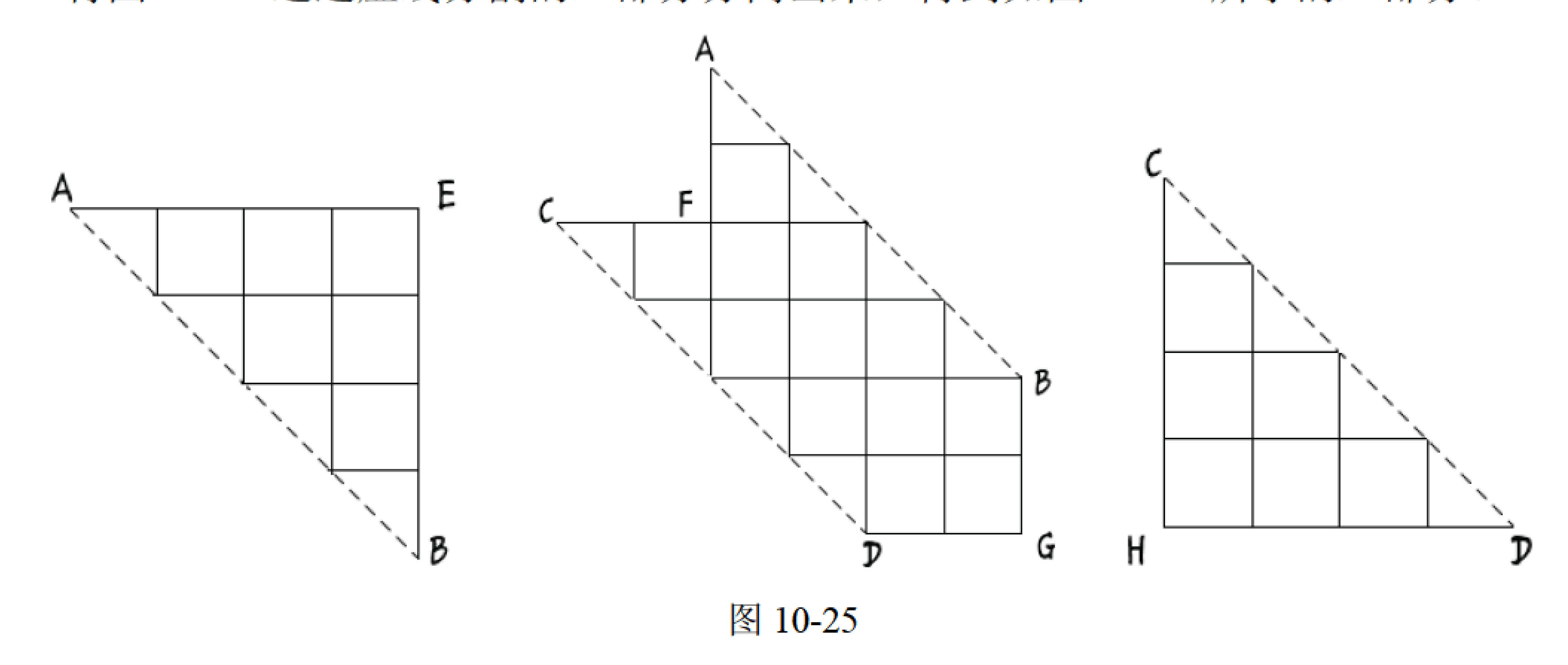


图 10-25

在图 10-25 中可以看到，虽然有 4 条边的边长为 $4\sqrt{2}$，但是，由这三个图形显然拼接不了一个正方形。因此，图 10-24 中设置的 CD 分割线不正确。

仔细观察图 10-23，左上角有一个边长为 2 的缺口，而经过 AB 线分割后，右下角也有一部分边长为 2。这时，可考虑将右下角边长为 2 的部分切割嵌入到左上角。因此，可考虑绘制如图 10-26 所示的一条分割线 CD。

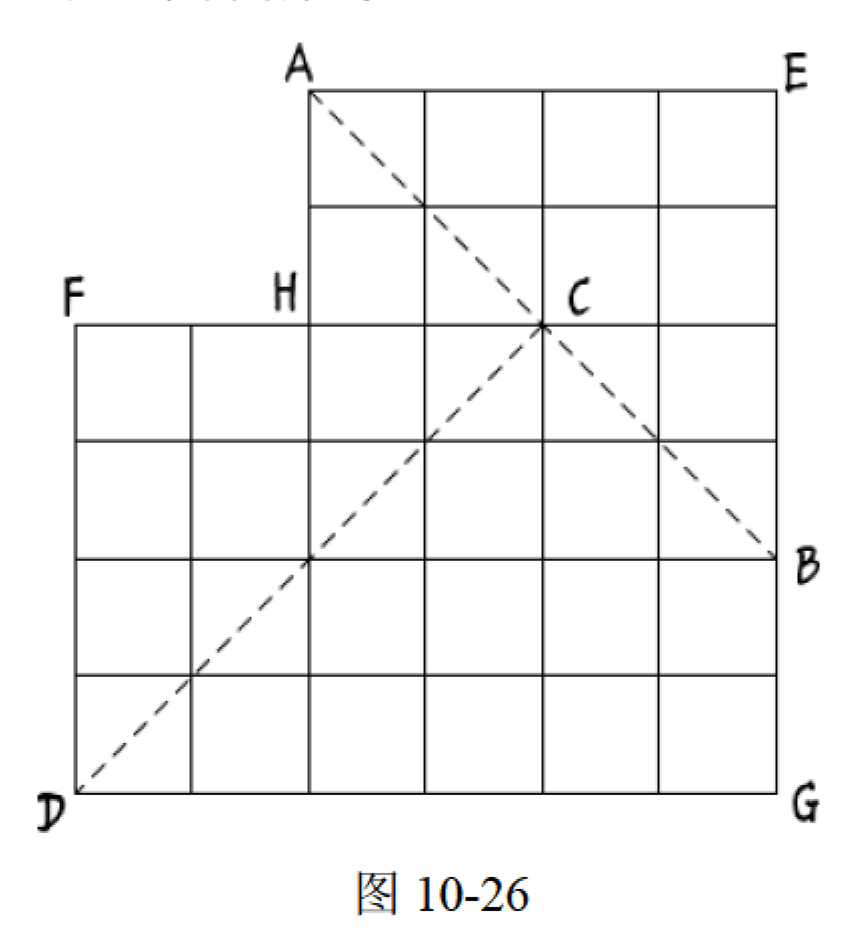


图 10-26

图 10-26 所示的两条分割线可将该图分割为如图 10-27 所示的三部分，感觉这三部分应该能拼接成一个正方形了。

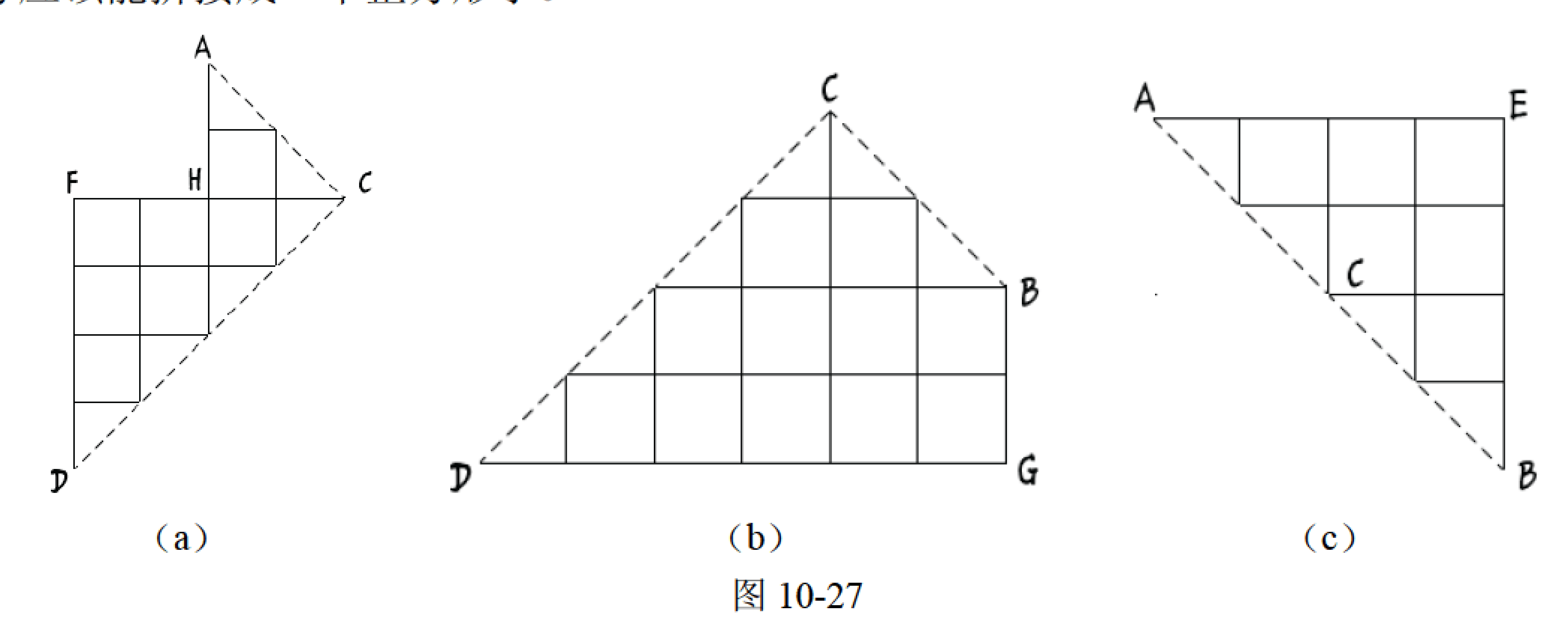


图 10-27

在图 10-27 所示的三个图中，将（b）图放在（a）图左上方，将（c）图放在前两个拼接图形的左下方，即可得到一个正方形，如图 10-28 所示。

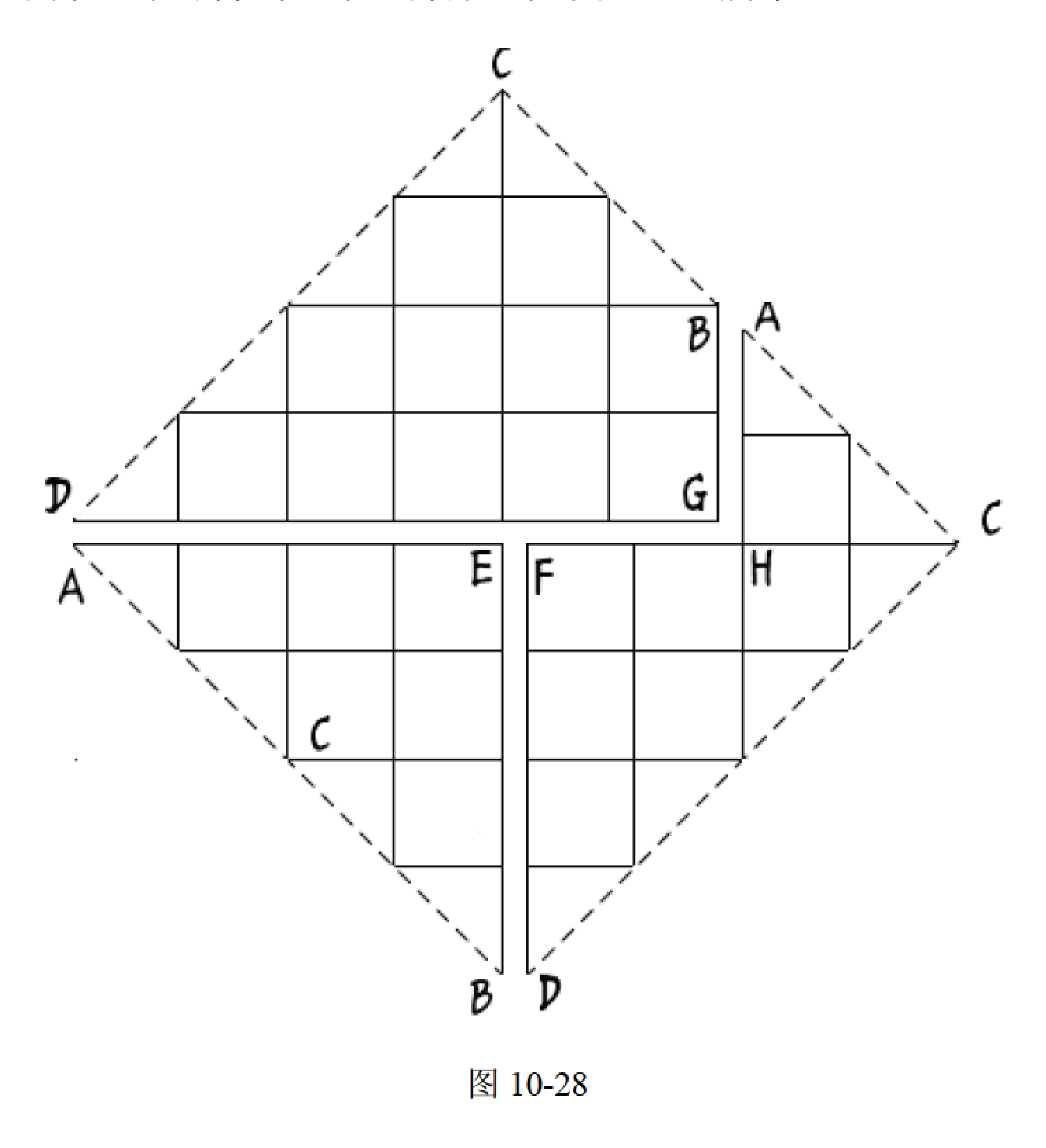


图 10-28

第 11 章　统 筹 规 划

统筹学是一门数学学科，但它在许多领域中都得到了应用。例如，在日常生活中有很多事情要去做时，如果能科学地统筹安排好先后顺序，将能够提高我们的工作效率。

11.1　认识统筹规划

听到“统筹规划”这个词语，很多人感觉是一门很高深的学问，因为我们经常在一些宏观经济、政策方针类的文件中看到或听到这个词。其实，我们的生活中随时都会用到“统筹规划”，这并不是什么高深的学问。

那么，什么是统筹规划？下面我们先来看一个古老的故事。

11.1.1　田忌赛马

齐国的大将田忌很喜欢赛马。有一回他和齐威王约定，进行一次比赛。

他们把各自的马分成上、中、下三等。比赛的时候，上等马对上等马，中等马对中等马，下等马对下等马。由于齐威王每个等级的马都比田忌的强一点，三场比试下来，田忌都失败了。田忌觉得很扫兴，垂头丧气地准备离开赛马场。

这时，田忌发现，他的好朋友孙膑也在人群里。孙膑招呼田忌过来，拍着他的肩膀，说：“从刚才的情形看，齐威王的马比你的马快不了多少呀……”

孙膑还没说完，田忌瞪了他一眼，说：“想不到你也来挖苦我！”

孙膑说：“我不是挖苦你，你再同他赛一次，我有办法让你取胜。”

田忌疑惑地看着孙膑：“你是说另换几匹马？”

孙膑摇摇头，说：“一匹也不用换。”

田忌没有信心地说：“那还不是照样输！”

孙膑胸有成竹地说：“你就照我的主意办吧。”

齐威王正在得意洋洋地夸耀自己的马，看见田忌和孙膑过来了，便讥讽田忌：“怎么，难道你还不服气？”

田忌说：“当然不服气，咱们再赛一次！”

齐威王轻蔑地说：“那就来吧！”

一声锣响，赛马又开始了。

孙膑让田忌先用下等马对齐威王的上等马，第一场输了。

接着进行第二场比赛。孙膑让田忌拿上等马对齐威王的中等马，胜了第二场。齐威王有点儿心慌了。

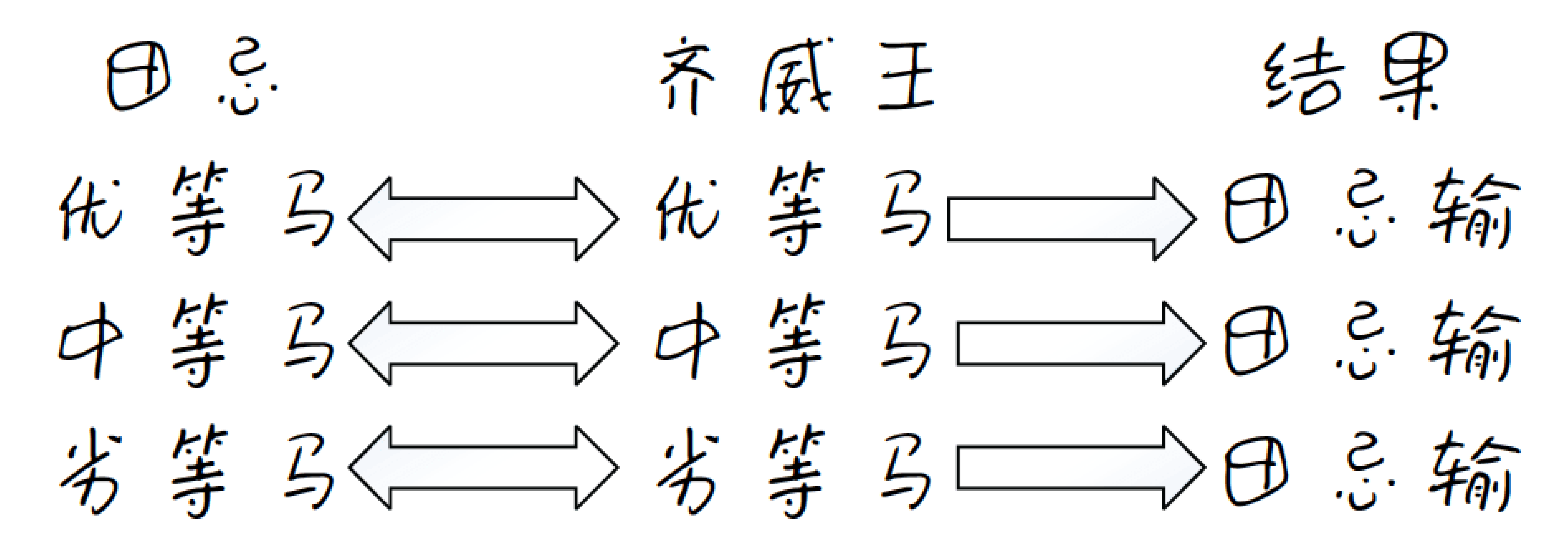
第三场，田忌拿中等马对齐威王的下等马，又胜了一场。这下，齐威王目瞪口呆了。

比赛结果，田忌胜两场输一场，赢了齐威王。

还是原来的马，只调换了一下出场顺序，就可以转败为胜。

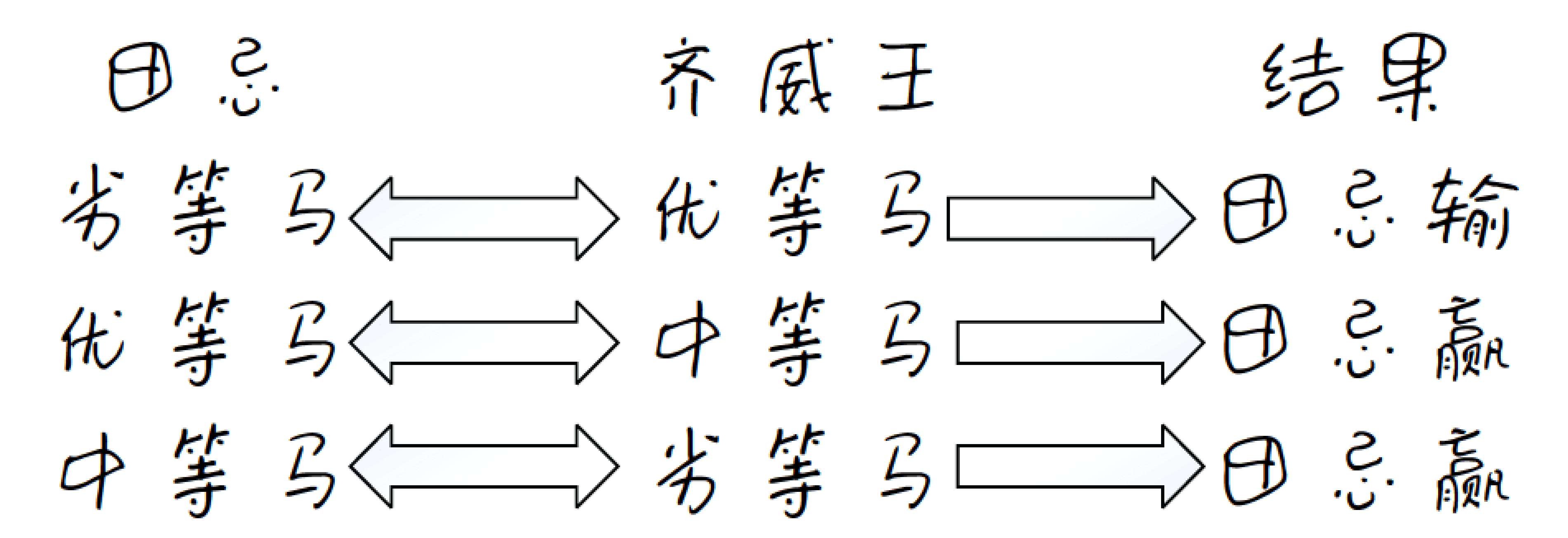
11.1.2　为什么会赢

首先，我们来看一下原方案的比赛对阵情况：



由于齐威王每个等级的马都比田忌对应等级的马强一点，因此三场比赛下来，田忌没赢一场。

同样的三匹马，孙膑只是改了一下出场顺序就赢得了比赛。我们看一下修改出场顺序后的对阵情况：



在修改后的对阵情况中，孙膑主动用己方最差的“劣等马”去与对方的“优等马”进行比赛，主动输一场。接下来用己方的“优等马”对阵对方的“中等马”，由于己方的“优等马”只比对方“优等马”略差一点，比对方的“中等马”要强很多，因此这种对阵肯定能赢；类似地，用己方的“中等马”对阵对方的“劣等马”也肯定能赢。最终获得两赢一输的成绩。

在这个故事中，孙膑提前对比赛进行了规划，主动输第一场，换取后两场的胜利。这就是统筹规划知识，也称为统筹学、运筹学。

统筹学的目的是：依据给定条件和目标，从众多方案中选择最佳方案。

如在田忌赛马中，双方派出参赛的马就有各种不同的出场顺序，要解决的问题就是如何统筹规划好出场顺序，使己方能获得最多的胜利场次。

可以看出，统筹学是利用数学来研究人力、物力的运用和筹划，使它们能发挥最大效率。它包含的内容非常广泛，例如，物资调运、场地设置、工作分配、排队、对策、实验最优等，每类问题都有特定的解法。

运筹学作为一门科学，要运用各种初等的和高等的数学知识及方法，但是其中分析问题的某些朴素的思想方法，如高效率优先的原则、调整比较的思想、尝试探索的方法等，都不需要高深的理论，具有小学文化知识就能掌握。

11.2 生活中的统筹规划

统筹学在我们的日常生活中运用非常普遍。在生活、工作安排等方面都可使用统筹规划相关知识来获取最佳方案，如减少工作时间，提高工作效率；降低物质占用，提高效益等。下面，我们来看一些常见的例子。

11.2.1 匆忙的早晨

现在的年轻人总是觉得早上时间不够用，又想多睡会，又要考虑上班不能迟到。每天早晨起床后又有很多事情要做，因此，总是弄得手忙脚乱，匆匆忙忙的出门上班。

例如，在工作日，小东早上 8 点半之前必须赶到公司打卡，尽管每天早上 7 点就起床了（还想多睡一会啊），但还是觉得时间不够，感觉很匆忙。小东每天早上起床后需要做以下事情：

- 收拾床铺及卧室，需要 5 分钟。
- 上卫生间，需要 10 分钟。
- 洗漱，需要 5 分钟。
- 早上自己做面条吃，共需要 20 分钟（烧开水要 15 分钟，下面条到出锅需要 5 分钟）。
- 接着吃早饭，需要 10 分钟。
- 用手机查看一下昨晚是否有重大新闻，需要 5 分钟。
- 乘公交车到公司，需要 20 分钟。

如果不改变小东的生活习惯，有没有什么办法能使他觉得时间很充足，甚至还可以多睡一会呢？

根据小东的生活习惯，可看到早上要做很多的事情，如果这些事情按顺序来安排，

可制作一个流程图如图 11-1 所示，框中为要做的事情，箭头指向下一件要做的事，箭头上的数字为前一件事需要花费的时间。

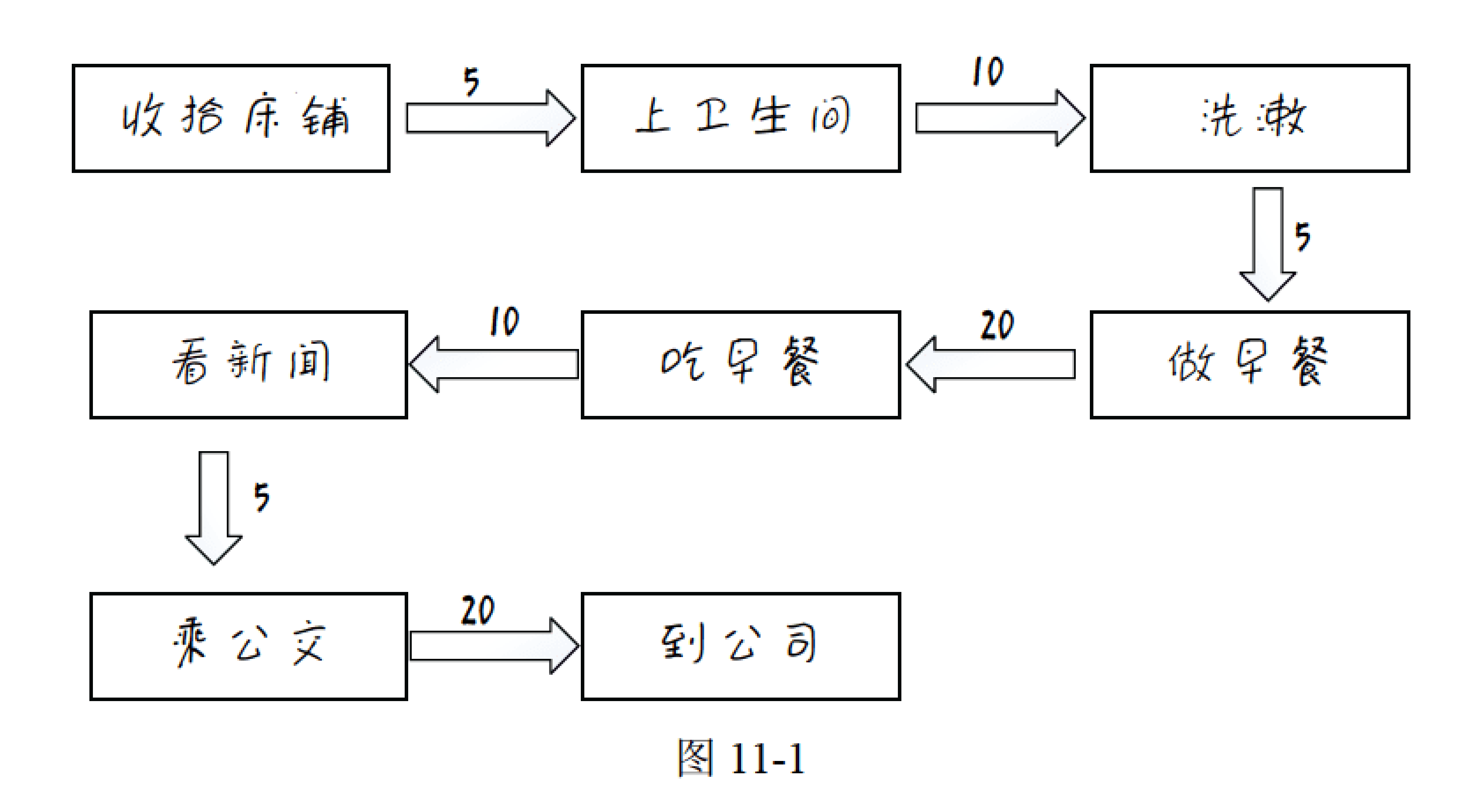


图 11-1

将图 11-1 所示箭头中的时间进行累加得到 75 分钟，即表示小东从早晨起床一直到公司，总共需要 75 分钟的时间。

有没有办法缩短总的时间呢？对图 11-1 中的各事项进行分析可以发现，在做某些事情时，其实不需要人全部参与。例如，在“做早餐”时，将水放在灶上烧着，由于将水烧开需要 15 分钟时间，而这 15 分钟时间里可以做其他的事情，这就是一种统筹规划。

下面，我们将小东早上要做的各事情进行分类，看看有哪些事情是可以同时做的，哪些事情必须有先后顺序。

其中，必须先“做早餐”，然后才能“吃早餐”、最后才能“乘公交”“到公司”，这些事情是有先后顺序的，而且这些事情的顺序不能打乱，如图 11-2 所示。从图中可看到完成这些事情共需要 50 分钟时间。

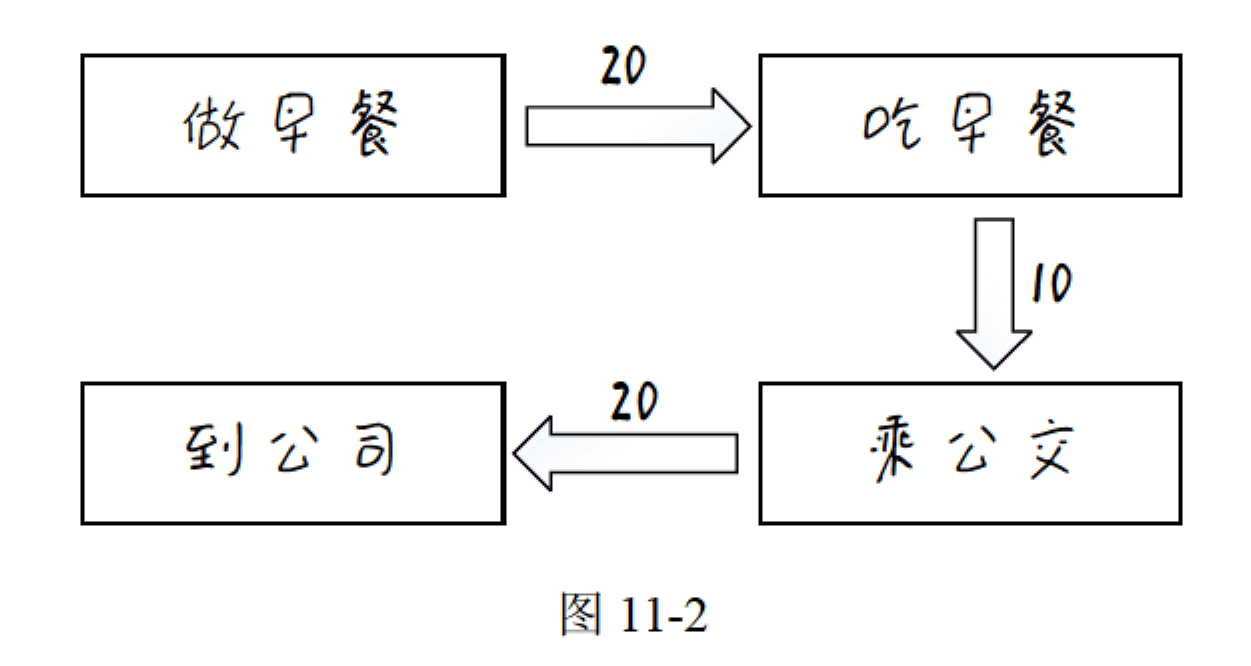


图 11-2

图 11-1 所列出的其他 4 件事情是没有先后顺序要求的，例如，可以先“上卫生间”，再“收拾床铺”，也可以先“收拾床铺”，再“上卫生间”。

对于没有先后顺序要求的事情，可以将其穿插到有先后顺序的事情中，与图 11-2 所示事情中不需人参与的事情同步进行。因此，可以将早上要做的事按图 11-3 所示方式进行。

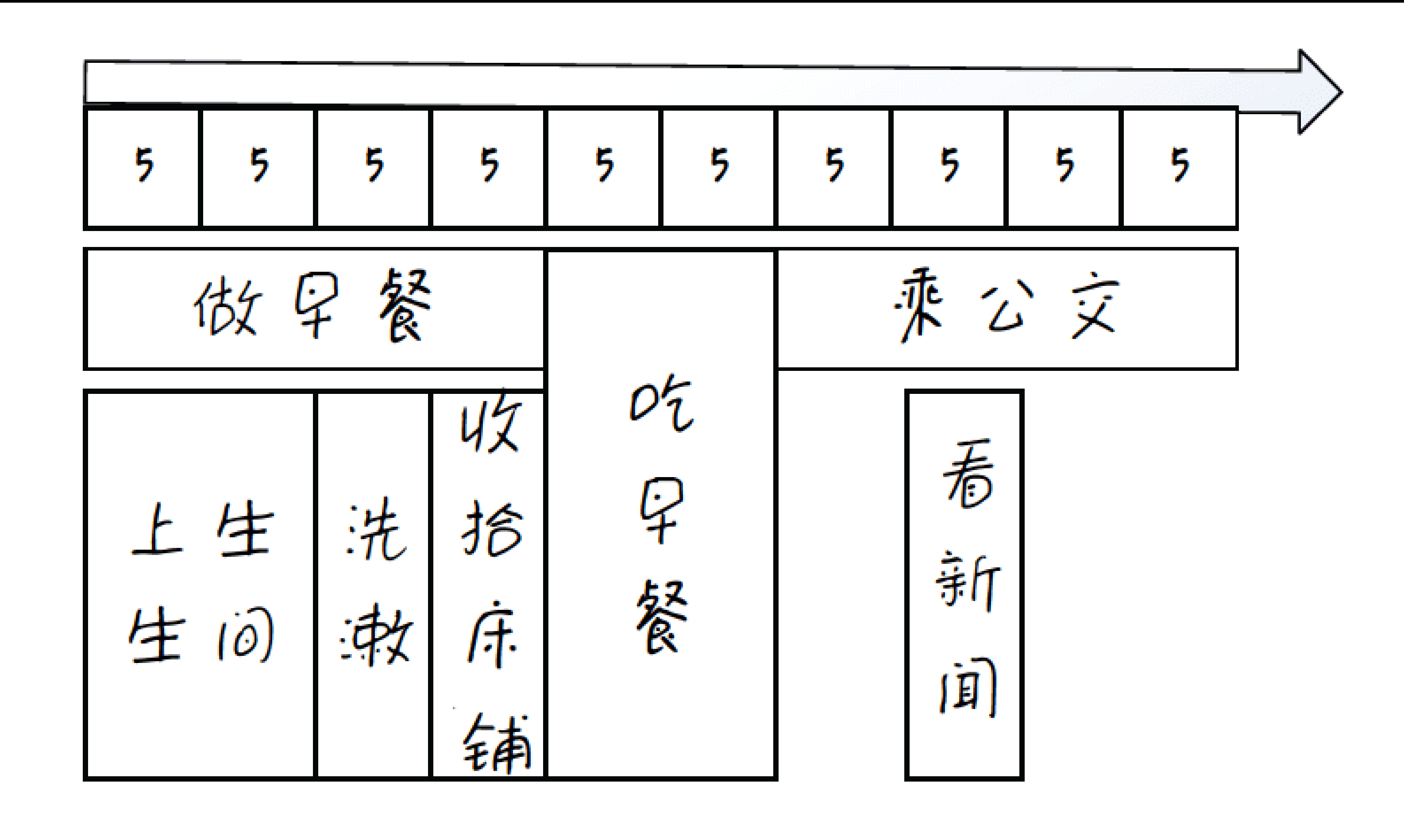


图 11-3

在图 11-3 中，上方的箭头表示时间流向，箭头下方每一个小方格表示 5 分钟时间，通过这个图可以很清晰地到看到每件事情所需的时间，以及做每件事情的时机。根据图 11-2 知道“做早餐”、“吃早餐”、“乘公交”这三件事是有时间顺序的，因此将这三件事情按时间顺序及每件事情所需时间长短绘制出来。接着就可安排无时间顺序的几件事情了。

根据图 11-3 可看出，小东早上起床第一件事就是烧水做早饭，然后在烧开水的过程中可“上卫生间”、“洗漱”。等“洗漱”完成后水也烧开了（15 分钟烧开水），就可下面条煮 5 分钟，而这 5 分钟又可“收拾床铺”。等早餐做好后，其他事情也做完了，接着就“吃早餐”，然后“乘公交”去上班，在公交上可用手机“看新闻”，因为公交需要 20 分钟时间。

通过以上的统筹安排，可将小东早上原来需要 75 分钟完成的事情缩短到 50 分钟。

11.2.2　如何节约运输成本

某仓储公司要将 57 吨货物从甲仓库转运到乙仓库，公司用于转运的车辆有两辆，1 号货车每次可转运 5 吨，2 号货车每次可转运 2 吨。现在已经知道 1 号货车从甲仓库跑到乙仓库及返空回甲仓库需消耗 10 升汽油，2 号货车从甲仓库跑到乙仓库及返空回甲仓库需消耗 5 升汽油。应该如何安排车辆进行转运，才能使转运成本最低？最低需要消耗多少升汽油？

这种问题很好解决。

要解决最低成本问题，首先应计算出每辆车转运时的吨位成本，即分别计算出 1 号货车转运时每吨的油耗是多少？2 号货车转运时每吨的油耗是多少？然后就尽量选择单位油耗少的车进行转运。这样，总体油耗量就最少，成本最低。

下面计算两辆车的吨位油耗。1 号车运输 5 吨，消耗 10 升汽油：

$$1号车吨油耗 = 10 \div 5 = 2 升/吨$$

2 号车运输 2 吨，消耗 5 升汽油：

$$2号车吨油耗 = 5 \div 2 = 2.5 升/吨$$

经过比较，1 号车每吨油耗比 2 号车低，因此，应尽量派 1 号车进行转运，如果 1 号车装不满时可再考虑选用 2 号车进行转运。

这里需要对 57 吨货物进行转运，1 号车每次只能转运 5 吨，则：

$$57 = 5 \times 11 + 2$$

可以看出，用 1 号车进行 11 次转运之后，还剩 2 吨货物。对于这 2 吨货物，若仍然用 1 号车进行转运，需消耗 10 升汽油。而 2 号车转运一次只消耗汽油 5 升，2 号车一次最多可转运 2 吨货物，因此这 2 吨货物用 2 号车转运可节省 5 升汽油。

一共需要消耗的汽油为：

$$11 \times 10 + 1 \times 5 = 115升$$

上面的解法是按最简单的优化思想——选择最低单位成本的方案进行的，其实，我们也可用代数算式来进行求解，具体过程如下。

设使用 1 号车转运 x 次，2 号车转运 y 次的运输成本最低，则可得下面的代数式：

$$5x + 2y = 57$$

以上算式中，$5x$ 表示 1 号车经过 x 转运，最多能转运的吨数，同样 $2y$ 表示 2 号车能转运的吨数。这个算式是假设正好两车都满载，分别经过 x、y 次转运，正好完成 57 吨货物的转运。

将以上代数式两边都乘以 2，并进行移项，可得：

$$10x = 114 - 4y$$

接下来计算总的油耗，可用以下代数式进行计算：

$$W = 10x + 5y$$

以上算式中，$10x$ 表示经过 x 次转运 1 号车消耗的汽油数，$5y$ 表示 2 号车消耗的汽

油数。以上两个算式中都有 $10x$（这下能理解为什么要将前面的算式两边乘 2 了吧），代入后可得以下算式：

$$W = 114 - 4y + 5y$$
$$= 114 + y$$

可以看出，转运货物总的油耗只与 2 号车的出车次数 y 有关，y 越小，总的油耗 W 也就越小。

根据前面的算式可知道，y 的最小值为 1（即 2 号车出车一次），因此，转运 57 吨货物总的油耗就为：

$$W = 114 + 1 = 115 \text{升}$$

可以看出，通过代数式方式计算得出的结果，与前面通过简单的优化思想（选择最低单位成本的方案）得到的结果是完全相同的。

11.3 著名的背包问题

背包问题是一个经典的动态规划求解问题。它既简单形象、容易理解，又在某种程度上能够揭示动态规划的本质。在很多地方都可以看到这类问题的描述，下面我们来讨论这个问题。

11.3.1 什么是背包问题

背包问题是一个求某种组合优化的问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。问题的名称来源于如何选择最合适的物品放置于给定背包中。

先来看一个背包问题的具体案例描述。

现在有一个背包，最多只能装重量 8 公斤的物品，如果要用该背包装如下水果，要求使背包中装的物品的价值最大，应该装下列哪些物品才能达到要求？最大价值为多少？

各水果的重量和价值如下：

- 苹果：5 公斤，40 元。
- 梨：2 公斤，12 元。

- 桃：1 公斤，7 元。
- 葡萄：1 公斤，8 元。
- 香蕉：6 公斤，48 元。

显然，我们没办法一下子就计算出哪种组合能让背包中的物品价值最大，只有通过不断地进行测试，并记录下不同物体搭配的价值，最后比较得出最大的价值。

要解决背包所装物品价值最大化的问题，可有很多种不同的方案。下面先用手工模拟一种方案。

第一步，可以从物品中任选一样装入背包。例如，选择“苹果”装入背包，则背包还剩 3 公斤重量的空余位置，这时，可记录装入物品的价值及总价值在如下表格中。

	装入物品	重量	剩余重量	价值	总价值
第 1 步	苹果	5	3	40	40

第二步，从剩余物品中再选一种，并判断该物品的重量是否超过背包的剩余重量，若未超过，可装入背包，并累加该物品的价值。例如，选择“梨”装入背包，将其重量、价值等记入以下表格中。

	装入物品	重量	剩余重量	价值	总价值
第 1 步	苹果	5	3	40	40
第 2 步	梨	2	1	12	52

第三步，重复第二步操作，从水果中选择“桃”装入背包，将其重量、价值等记入以下表格中。

	装入物品	重量	剩余重量	价值	总价值
第 1 步	苹果	5	3	40	40
第 2 步	梨	2	1	12	52
第 3 步	桃	1	0	7	59

第四步，当背包已无法装下新的物品时，记下这次试装时背包所装物品的总价值。

接下来，从背包中拿出最后装入的物品，然后选择其他未装入背包的物品进行测试。例如，从上表中将“桃”拿出，重新将“葡萄”装入背包，将其重量、价值等记入以下表格中。

	装入物品	重量	剩余重量	价值	总价值
第1步	苹果	5	3	40	40
第2步	梨	2	1	12	52
第3步	葡萄	1	0	8	60

可以看出，背包中装入“葡萄”比“梨”的总价值要高。这时，就将目前的最高价值记录下来。

接着，再从背包中取出最后装入的“葡萄”，试着装入“香蕉”，可是背包剩余重量为1公斤，没办法装入6公斤重的“香蕉”。接着将第2步装入背包的“梨”取出，背包剩余重量为3公斤，仍然无法装入“香蕉”。继续将第1步装入背包的“苹果”取出，背包剩余重量为8公斤。这样，就可将“香蕉”装入背包了，这时背包中只有这一件物品，其重量、价值等数据如下表。

	装入物品	重量	剩余重量	价值	总价值
第1步	香蕉	6	2	48	48

重复前面的步骤，将“梨”装入背包，记录重量、价值数据到如下表格中。

	装入物品	重量	剩余重量	价值	总价值
第1步	香蕉	6	2	48	48
第2步	梨	2	0	12	60

由于背包已没有剩余重量了，记录背包总价值。

重复前面的步骤，从背包中取出最后装入的物品，再测试装入其他物品。这样不断循环，直到将各种物品组合都测试完成，找出价值最高的那一次试装入即可。

看着这样重复不断试装入的步骤，是不是感觉很繁琐？在反复的装入、取出操作中，常常还会出现重复操作情况。有什么好的解决办法呢？对于重复的、相似的操作，最简单的方法就是编写程序，让计算机帮我们来完成。

11.3.2　用递归程序解决背包问题

可通过递归方法求解背包问题，递归方法解背包问题的流程如图11-4所示。

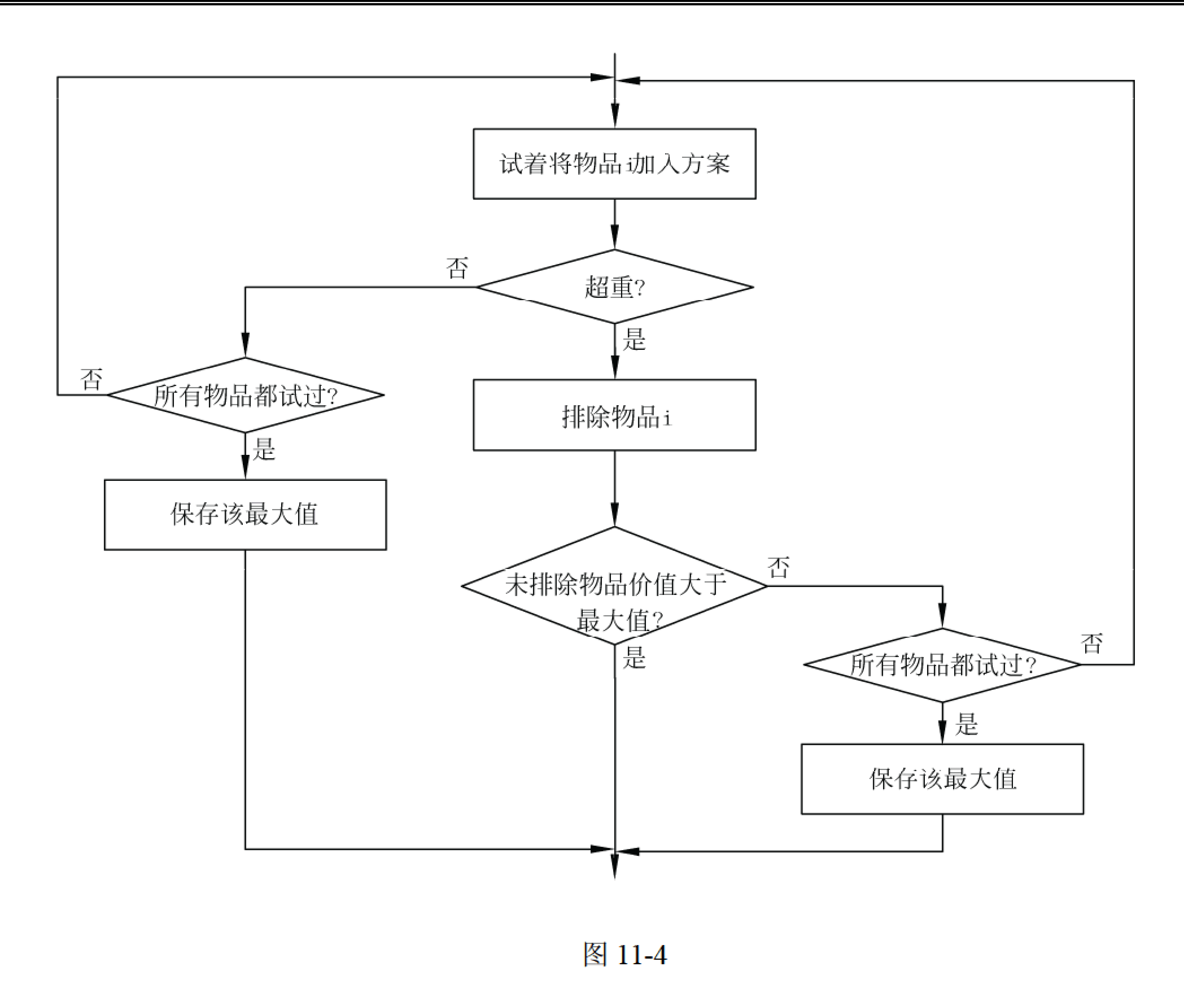


图 11-4

如图 11-4 所示，首先将物品 i 试着加入背包中，接着程序判断背包中装入物品 i 后是否超重，如果没有超重，则继续装入下一个物品；如果已超重，则将该物品排除在本次装入方案之外，并判断排除当前物品后，所有未排除物品的价值是否小于已有最大值，若是，则不必再测试后续物品。

根据图 11-4 所示流程图，可编写出用递归方式解决背包问题的程序，具体代码如下：

```
#include <stdio.h>

typedef struct goods    //定义结构
{
     double *value;     //保存各物品价值（数组）
     double *weight;    //保存各物品重量（数组）
     char *select;      //保存各物品是否装入背包（数组）
     int num;           //可装入背包的物品数量
     double limitw;     //背包最大重量
}GOODS;

double maxvalue;        //装入背包物品的最大值
double totalvalue;      //所有物品的总价值
char *select1;          //临时数组

void backpack(GOODS *g, int i, double tw, double tv)
```

```
//参数说明
//g 传入要处理物品结构指针，
//i 需要试装入物品的序号
//tw 当前背包已经达到的总重量
//tv 所有未装入背包的物品的总价值
{
   int k;
   if (tw + g->weight[i] <= g->limitw)  //试装入物品 i，未超重
   {
     select1[i] = 1;                     //选中第 i 个物品装入背包
     if (i < g->num - 1)                 //若物品 i 不是最后一个物品
       backpack(g, i + 1, tw + g->weight[i], tv);
                                         //递归调用，继续试装入下一物品
     else                                //若已到最后一个物品
     {
       for (k = 0; k < g->num; ++k)      //将状态标志复制到 select 数组中
         g->select[k] = select1[k];
       maxvalue = tv;                    //保存当前背包中的最大价值
     }
   }
   select1[i] = 0;                       //从背包中取出物品 i
   if (tv - g->value[i] > maxvalue)
   //若未装入背包的物品总价值减去物品 i 的价值还大于 maxvalue
   //说明还可以继续向背包中添加物品
   {
     if (i < g->num - 1)                 //若物品 i 不是最后一个物品
       backpack(g, i + 1, tw, tv - g->value[i]);//递归调用，继续装入下一物品
     else                                //若已到最后一个物品
     {
       for (k = 0; k < g->num; ++k)      //将状态标志复制到 select 数组中
         g->select[k] = select1[k];
       maxvalue = tv - g->value[i]; /    //保存当前背包中的最大价值
     }
   }
}

int main()
{
   double sumweight;                //保存装入背包物品总价值
   GOODS g;
   int i;

   printf("背包最大重量:");
   scanf("%lf",&g.limitw);          //输入背包最大重量
   printf("可选物品数量:");
   scanf("%d",&g.num);              //输入物品数量
   if(!(g.value = (double *)malloc(sizeof(double)*g.num)))
                                    //分配内存保存物品价值
   {
```

```
        printf("内存分配失败\n");
        exit(0);
    }
    if(!(g.weight = (double *)malloc(sizeof(double)*g.num)))
                                        //分配内存保存物品的重量
    {
        printf("内存分配失败\n");
        exit(0);
    }
    if(!(g.select = (char *)malloc(sizeof(char)*g.num)))
                                        //分配内存保存物品的重量
    {
        printf("内存分配失败\n");
        exit(0);
    }
    if(!(select1 = (char *)malloc(sizeof(char)*g.num)))
                                        //分配内存保存物品的重量
    {
        printf("内存分配失败\n");
        exit(0);
    }

    totalvalue=0;
    for (i = 0; i < g.num; i++)  //输入各物品的重量和价值
    {
      printf("输入第%d 号物品的重量和价值:",i + 1);
      scanf("%lf%lf",&g.weight[i],&g.value[i]);
      totalvalue+=g.value[i];     //统计所有物品的价值总和
    }

    printf("\n 背包最大能装的重量为:%.2f\n\n",g.limitw);
    for (i = 0; i < g.num; i++)
      printf(" 第 %d 号 物 品 重 :%.2f, 价 值 :%.2f\n", i + 1, g.weight[i],
g.value[i]);

    for (i = 0; i < g.num; i++)  //初始设各物品都没装入背包
        select1[i]=0;

    maxvalue=0;  //装入背包物品的总价值
    backpack(&g,0,0.0,totalvalue);  //调用函数将第 1 个物品装入背包

    sumweight=0;
    printf("\n 可将以下物品装入背包,使背包装的物品价值最大:\n");
    for (i = 0; i < g.num; ++i)
      if (g.select[i])            //若装入背包
      {
        printf("第%d 号物品,重量:%.2f,价值:%.2f\n", i + 1, g.weight[i],
g.value[i]);
        sumweight+=g.weight[i];
```

```
    }
    printf("\n 总重量为: %.2f,总价值为:%.2f\n", sumweight, maxvalue );

    getch();
    return 0;
}
```

以上程序代码比较长，不过在关键代码处都有详尽的注释。为了使程序具有一定的通用性，在主函数 main()中，采用动态分配内存的方式，让用户输入物品数量、各物品的重量和价值、背包的最大重量等参数。

编译执行以上程序，按提示输入背包最大重量、物品数量及各物品的重量和价值，程序就可找出使背包中物品价值最大的组合。我们将前面例子中的数据输入，就可得到图 11-5 所示的计算结果。

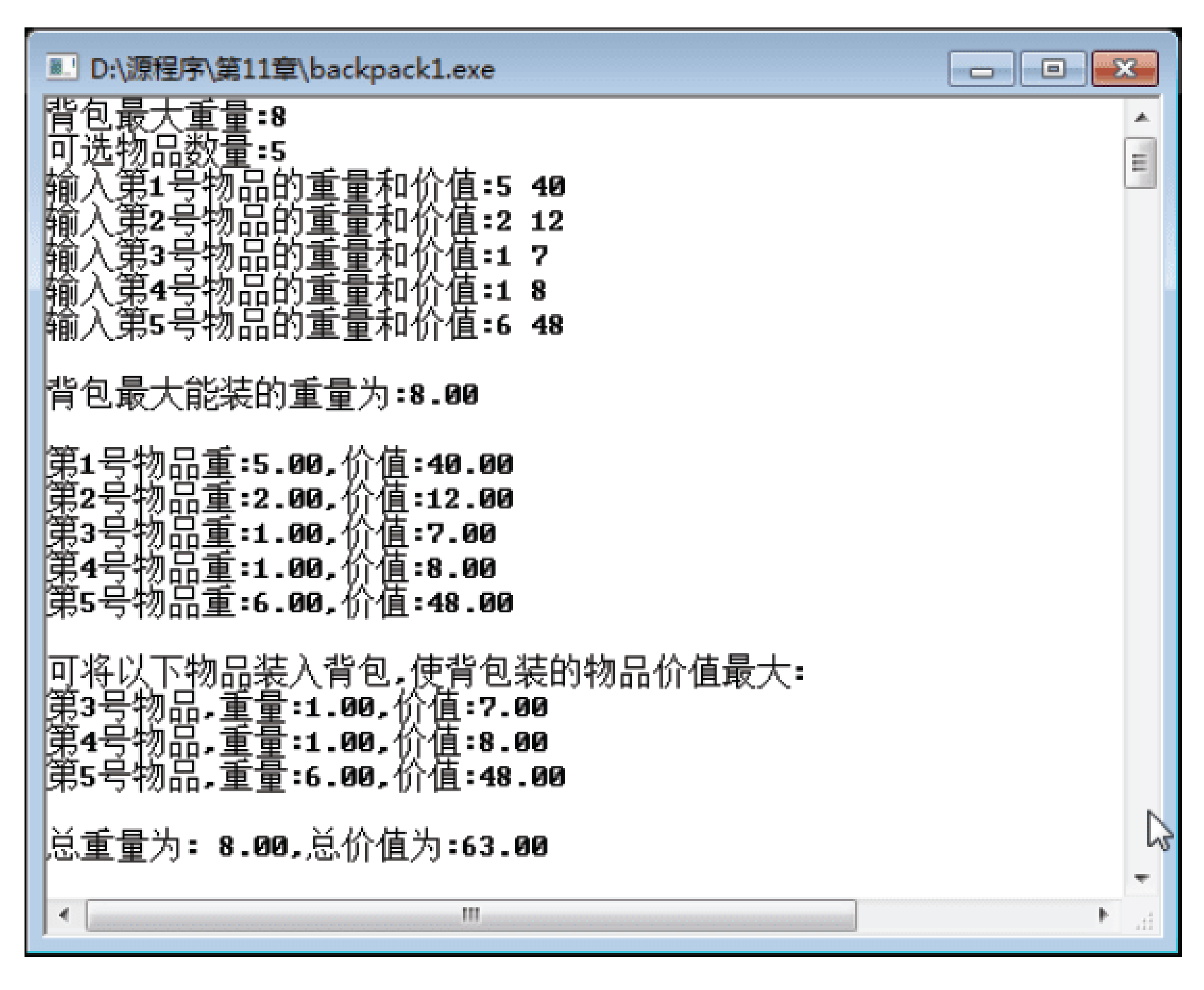


图 11-5

11.3.3 用穷举法解决背包问题

对于背包问题，也可用穷举法来解决。将可装入背包的物品通过不同的组合，试算其重量是否超过限制重量，若该组合未超过限制重量，则累加该方案中各物品的价值，得到一个总价值，再用该总价值和已有方案的最高价值进行比较，若该方案的物品价值更大，则保存该方案。

通过穷举将所有可能的组合都测试过之后，得到的就是最优的方案。

接下来就需要考虑用什么方法来得到由不同物品组成方案的组合。最简单的是：对于 n 个物品使用 n 层循环，这样就可以得到各种不同的组合。但是，对于背包问题，其物品的数量是不确定的。

对于每个物品，在生成的组合中有两种可能：一是加入背包，一是排除在背包之外。对于这种由多个物品组成，每个物品有两种可能的情况，可以使用二进制数来进行模拟。对于 n 个物品，就可用 n 位二进制数来进行模拟，当某位为 1 时，表示对应物品加入背

包，为 0 时，表示不将该物品加入背包。例如，对于前面例子中的 5 种水果，就可用 5 个二进制位来表示某一种装入方案。图 11-6 表示一种方案，其中第 1、4、5 位为 1，表示将第 1、4、5 种水果装入背包，而将第 2、3 种水果不装入背包。

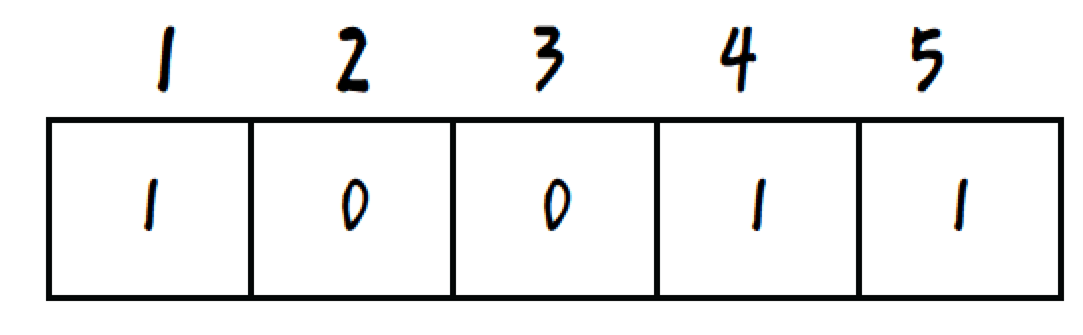


图 11-6

有了这种方案之后，接下来是二进制的表示问题。为了简化程序，可使用字符数组来保存二进制数据，一个数组元素只保存一位二进制数，也就是说，一个字符数组元素只保存 0 或 1 这两个数值之一，这样就可方便地运算。

使用二进制来穷举所有组合，首先将表示二进制数的数组全部清 0，然后向该数组的低位元素逐步进行加 1 操作，当每一个数组元素都为 1 时，就表示穷举了所有可能。

根据以上分析，编写用穷举法进行背包问题求解的程序，具体实现代码如下：

```
#include <stdio.h>

typedef struct goods
{
    double *value;                          //各物品的价值（数组）
    double *weight;                         //各物品的重量（数组）
    int num;                                //物品数量
    int limitw;                             //背包限制重量
}GOODS;

void backpack(GOODS *g,char select[])
//参数说明：
//g 指向保存物品信息的结构
//select 用来返回物品的装入状态
{
   int i,flag;
   char *select1;                           //保存物品装入状态
   double maxvalue = 0;
   double tw;                               //装入背包物品总重量
   double tv;                               //装入背包物品总价值
   if(!(select1 = (char *)malloc(sizeof(char) * g->num)))
   {
      printf("内存分配失败\n");
      exit(0);
   }

   for (i = 0; i < g->num; i++)             //将数组清空，表示全部未装入背包
      select1[i] = 0;
```

```
    while(binadd(select1, g->num) == 0)  //进行一次二进制加法运算
    {
        tw = 0;
        tv = 0;
        flag = 1;

        for (i = 0; i < g->num; i++)      //根据物品装入背包的状态进行试算
        {
            if (select1[i] == 1)          //若将该物品装入背包
            {
                tw += g->weight[i];       //累加装入物品的重量
                tv += g->value[i];        //累加装入物品的价值
                if (tw > g->limitw)       //若重量超过限制
                {
                    flag = 0;
                    break;                //退出本次试装入
                }
            }
        }

        if(flag && maxvalue < tv)
                    //若前包中物品重量未超过限制，并且本次累加价值大于已有最大值
        {
            maxvalue = tv;                //保存最大值
            for(i = 0; i < g->num; i++)   //保存物品状态
                select[i] = select1[i];
        }
    }
}

int binadd(char select1[],int n)          //二进制运算
{
    int i,carry=0;
    select1[0] += 1;                  //低位加 1
    for (i = 0; i < n; i++)
    {
        select1[i] += carry;          //加上进位
        carry = select1[i] /2;        //计算进位
        select1[i] %= 2;              //保留 0 或 1;
        if (carry==0)
            return 0;
    }
    return carry;
}

int main()
{
   double sumweight,maxvalue;         //用来保存阶段最优价值
   char *select;                      //保存最大价值时各物品的装入状态
```

```
   GOODS g;
   int i;

   printf("背包最大重量:");
   scanf("%d",&g.limitw);
   printf("可选物品数量:");
   scanf("%d",&g.num);

   if(!(g.value = (double *)malloc(sizeof(double)*g.num)))
                                    //分配内存保存物品价值
   {
      printf("内存分配失败\n");
      exit(0);
   }

   if(!(g.weight = (double *)malloc(sizeof(double)*g.num)))
                                    //分配内存保存物品的重量
   {
      printf("内存分配失败\n");
      exit(0);
   }

   if(!(select = (char *)malloc(sizeof(char)*g.num)))
                                    //分配内存保存物品状态
   {
      printf("内存分配失败\n");
      exit(0);
   }

   for(i=0;i<g.num;i++)
   {
     printf("输入第%d 号物品的重量和价值:",i + 1);
     scanf("%lf%lf",&g.weight[i],&g.value[i]);
   }

   printf("\n 背包最大能装的重量为:%.2f\n\n",g.limitw);
   for (i = 0; i < g.num; i++)
     printf(" 第 %d  号 物 品 重 :%.2f, 价 值 :%.2f\n",  i  +  1,  g.weight[i],
g.value[i]);

   backpack(&g,select);

   sumweight=0;
   maxvalue=0;
   printf("\n 可将以下物品装入背包,使背包装的物品价值最大:\n");
   for (i = 0; i < g.num; ++i)
     if (select[i])      //装入背包
     {
       printf("第%d 号物品,重量:%.2f,价值:%.2f\n", i + 1, g.weight[i],
```

```
g.value[i]);
      sumweight+=g.weight[i];
      maxvalue+=g.value[i];
     }
  printf("\n总重量为: %.2f,总价值为:%.2f\n", sumweight, maxvalue );

  getch();
  return 0;
}
```

编译执行以上程序，可看到与用递归法求解背包问题的程序界面完全相同，输入相同的参数（包括背包最大重量、物品数量及各物品的重量和价值等）后，程序就可找出使背包中物品价值最大的组合。如果输入图 11-5 所示参数，得到的结果也与图 11-5 相同。